Journal of Research in Science and Engineering (JRSE)
ISSN: 1656-1996 Volume-7, Issue-2, February 2025
DOI: 10.53469/jrse.2025.07(02).03

# Implementing Micro Front End Architecture: Vue.Js' Diversified Strategy

**Muhammed Rafi**

**Abstract:** ***This paper presents an in - depth exploration of micro - frontend architectures, a burgeoning methodology designed to address the complexities and scalability challenges inherent in large - scale web application development. As digital projects expand, they often become unwieldy, with increased build times, extensive unit testing requirements, and larger team sizes, all of which contribute to maintenance and collaboration difficulties. This research argues for the subdivision of monolithic applications into smaller, more manageable projects, facilitated by micro - frontends, to enhance team performance and accelerate product delivery to end - users. However, the integration of numerous features within a single application can complicate the division into smaller projects and teams. Micro - frontends emerge as a strategic solution to this dilemma, not as a framework or library, but as an architectural approach that divides large applications into manageable segments. These segments, or micro - frontends, are then orchestrated to appear as a cohesive application to the end - user. The paper outlines eight distinct strategies for implementing micro - frontends, regardless of the underlying technology: Webpack Module Federation, Iframes, NGINX, Web Components, React Component Libraries, Monorepos, Utilization of Frameworks, and Custom Orchestrators. Through a high - level architectural overview, this study sets the stage for a series of detailed analyses on each approach, aimed at providing a comprehensive framework for deploying micro - frontend architectures effectively.***

**Keywords:** Vue. js, Micro - Frontends, Software Architecture, Programming, Software Development

## 1. Introduction

Through my extended research and practical experience with micro - frontends, I've amassed a considerable body of knowledge that I am eager to disseminate. The challenge of managing and collaborating on increasingly large projects is a notable concern within the development community. As projects scale, they often encounter heightened build times, expanded unit tests, and growing team sizes, culminating in a complexity that becomes nearly unmanageable.

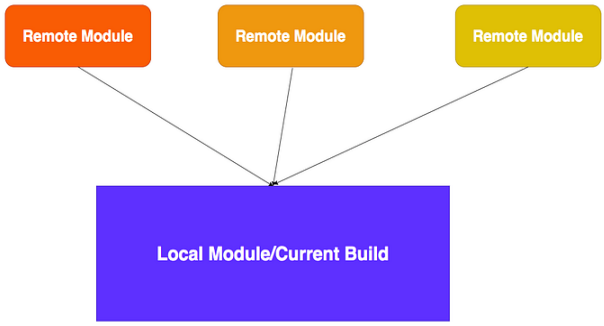
The argument for segmenting larger projects into smaller, more manageable units is compelling, particularly from the perspective of enhancing team performance and expediting the delivery of products to end - users. However, the reality is that the complexity of modern applications, replete with myriad features, frequently renders the division into smaller teams and projects unfeasible. While the idea of assigning separate teams to individual features may appear as a solution, the logistics involved in team management, feature integration, and conflict resolution present significant challenges. Micro - frontends offer a promising avenue to circumvent these obstacles.

Micro - frontends should not be misconstrued as merely another framework or library. Instead, they represent a sophisticated architectural methodology that enables the decomposition of large, cumbersome applications into smaller, more manageable components. These components can then be orchestrated in such a manner that they are presented seamlessly as a unified application to the end - user. My research elucidates eight distinctive strategies for the implementation of micro - frontends, transcending technological boundaries. These strategies include: Webpack Module Federation, Iframes, deployment via NGINX, Web Components, React Component Libraries, Monorepos, leveraging existing Frameworks, and the development of Custom Orchestrators. This paper will provide a preliminary overview of the high - level architecture of these strategies, setting the groundwork for a series of detailed follow - up discussions dedicated to the comprehensive implementation of each approach.

- ***Webpack Module Federation***
- ***Iframes***
- ***Through NGINX***
- ***Web Components***
- ***React Component Libraries***
- ***Monorepos***
- ***Customized Orchestrator***

### 1) Webpack Module Federation

The architecture of a singular application should be composed of numerous independent builds, each functioning as a standalone unit. This design principle ensures that these builds operate without interdependencies, allowing for their development and deployment to proceed autonomously. Such a structure not only facilitates parallel development streams but also enhances the flexibility and scalability of the deployment process, enabling updates and modifications to be made to one part of the application without necessitating adjustments to others.




**Webpack Module Federation**

In the context of modular application architecture, there exist two categories of modules: local and remote. Local modules refer to the components that are part of the current build or application in development, essentially constituting the

immediate codebase. On the other hand, remote modules are external components that must be imported into the current application or build, originating from separate development environments or builds.

Each build functions both as a self - contained unit and as a consumer of other builds, effectively adopting the role of a container. This dual capability enables any given build to access and utilize functionalities from other modules by importing them from their respective containers. Furthermore, the architecture supports the nesting of containers, allowing for a hierarchical organization where containers can incorporate modules from other nested containers, thereby fostering a highly flexible and interconnected module system.

Moreover, this model accommodates circular dependencies among containers, wherein two or more containers rely on each other's functionalities. Such a structure ensures a versatile and dynamic interaction between different parts of the application, enabling developers to create complex and highly integrated software solutions.

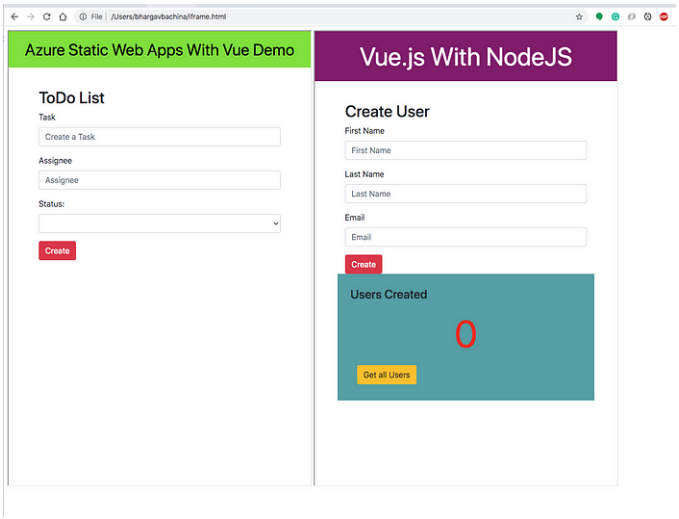
Module Federation (https: //webpack. js. org/concepts/module - federation/)

## 2) IFRAMES

Iframes represent a feature in HTML that allows the incorporation of one HTML document within another. This embedding is facilitated through the use of the iframe tag, which specifies the source document to be displayed within the confines of the parent document. An iframe can host any content chosen by the developer, and its display is governed by the dimensions—width and height—assigned to the frame within the parent document. This flexibility makes iframes a powerful tool for integrating varied content, such as videos, maps, or even other web pages, directly into a web page's layout.

- Iframe example (https: //gist. github. com/bbachi/37e1e1eb3019d9e50cd11ca8c533d823#file - iframe - html)

In the provided example, two separate React applications are being executed simultaneously on ports 3000 and 3001. These applications serve as the content sources for the iframes specified in the document mentioned previously. Upon loading this document in a web browser, the result is the display of both applications side by side within the same browser window, illustrating a straightforward method of content integration via iframes. While this instance serves as a basic demonstration of the concept, a more advanced and detailed example will be presented in forthcoming posts, aiming to further explore and elucidate the potential of iframe integration within web development.

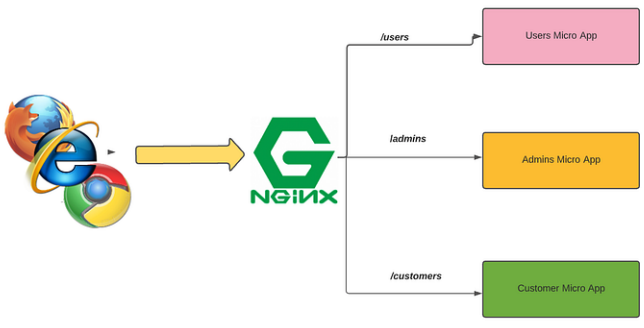



**Micro Frontends with Iframes**

This strategy is particularly advantageous for projects designed to consolidate all functionalities within a single page, eliminating the need for navigation between different pages. Communication between various components or functionalities is facilitated through the Window object, allowing for seamless interaction within the unified page environment. Such an approach ensures that users can access and interact with multiple features without the need to navigate away, providing a streamlined and cohesive user experience.

## 3) THROUGH NGINX

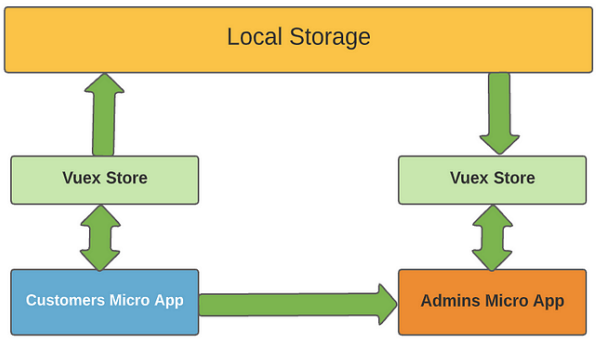
The described methodology may not be the most effective when the project involves complex navigation and routing mechanisms. While it is possible to implement this strategy, it necessitates the creation of an additional "shell" project to handle navigation intricacies. In such cases, NGINX can serve a dual role as both a web server and a reverse proxy, managing the delivery of static content efficiently. Utilizing NGINX's capabilities, one can configure routing to direct users to the correct micro - application based on the context path. For instance, in the illustrated setup, an NGINX web server acts as an intermediary, directing traffic to specific micro frontends according to the defined routing rules—such as directing requests for "/users" to the Micro Users app, and requests for "/customers" to the Micro Customers app. This arrangement allows for a modular, scalable approach to managing different segments of a web application, each serving a distinct function or serving different user bases, while maintaining a cohesive overall user experience.



**NGINX routing different apps based on context path**
Here is the sample NGINX configuration file. We define block directive location for each Micro Frontend and load appropriate app based on the location path or context root.

nginx. conf (https: //gist. github. com/bbachi/698a8512667b3d3fa6ec2f0034e8f14b#file - nginx - conf)

This strategy is particularly well - suited for projects that require navigation or routing and are structured as a collection of multiple applications, each corresponding to distinct features. In such setups, communication and data exchange between the various applications are facilitated with a state management tool like Vuex, alongside local storage mechanisms. As users transition from one application to another within this ecosystem, the state of the application—including user interactions, preferences, and data—is preserved and conveyed through these tools. This ensures a seamless user experience, with the application maintaining continuity and context across navigational transitions, as depicted in the accompanying diagram. Such an architecture enables the effective compartmentalization of features into discrete applications, while still providing a unified and cohesive interface for the end - user.




**Communication between Micro Frontends**

A notable drawback of this modular approach is the occurrence of a page refresh each time there is a transition between applications. This can interrupt the fluidity of the user experience, as each app switch involves reloading the webpage. To mitigate this and ensure a consistent user interface across the different applications, it's common to replicate shared components—like headers and footers—within each app. By doing so, even though the page refreshes, the overall layout remains familiar to the user, helping to maintain a sense of continuity and cohesion throughout the navigation process. This strategy helps in balancing the modular benefits of the approach with the need for a stable and uniform user experience.

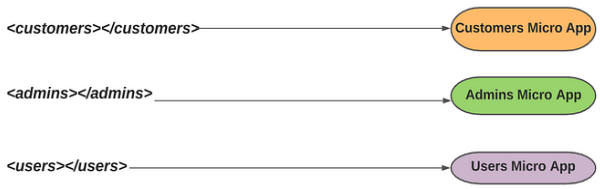
**4) Web Components**
Web components represent a synergistic integration of various technologies designed to foster the creation of reusable and modular elements within web development. Central to the ethos of the DRY (Don't Repeat Yourself) principle, web components aim to streamline development processes by eliminating redundant code. This is achieved through a trio of core technologies:

**Custom Elements:** These enable developers to define and use new types of HTML elements, extending the language to include custom functionality and styling. Custom elements can be placed anywhere on a web page, offering a high degree of flexibility in design and functionality.

**Shadow DOM:** This technology provides a means for encapsulating a piece of the document tree within a separate DOM, distinct from the main document DOM. This isolation allows developers to write styles and scripts that do not conflict with the rest of the web page, promoting a modular architecture.

**HTML Templates:** HTML templates facilitate the definition of markup structures that are meant to be used repeatedly. Using the <template> tag, developers can create chunks of HTML that can be cloned and inserted into the document as needed, without being rendered until explicitly invoked.

Together, these technologies empower developers to craft sophisticated, maintainable, and reusable web components, significantly enhancing the efficiency and quality of web development projects. For further details on web components and their implementation, additional resources and documentation are readily available to explore.




**Micro Frontends with web components**

Here is an example of a custom component. With this approach, we can convert each micro app into a custom component and place it accordingly on the page.

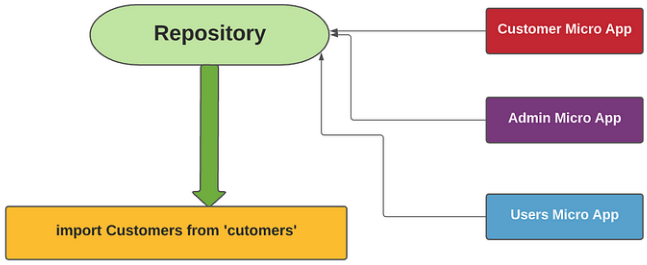
<my - message message="Hello, How are you!!"></my - message>

All the browsers don’t support these web components. You need to add polyfills for unsupported versions.

**5) VUE Component Libraries**
Within this methodology, each micro - frontend is developed as a Vue. js library, which is then packaged and distributed as a node module into a private repository. This encapsulation strategy allows for the modularized components to be easily shared and reused across different parts of the application or even across multiple projects. To integrate these micro - frontends into the main application, a "shell" application is employed. This shell application is designed to dynamically import or lazily load (in the case of Angular projects) the necessary micro - frontend libraries from the private repository as required, enhancing performance and reducing initial load times.

Illustrated in the diagram below, the concept is applied to three separate micro - applications. Each of these applications is transformed into a library, subsequently stored in a repository. This setup not only streamlines the development process by promoting reusability and modularity but also

facilitates a more efficient loading strategy. By adopting lazy loading or dynamic imports, the main application can request and load these micro - frontends on - demand, ensuring that users experience faster initial access times and that resources are consumed more judiciously. This approach exemplifies a modern architectural pattern that leverages the flexibility and scalability of micro - frontends within a cohesive ecosystem.




Micro Frontends with Vue Libraries

We can push our project as a node module with this commandnpm push app1We can import these libraries with dynamic imports as below.

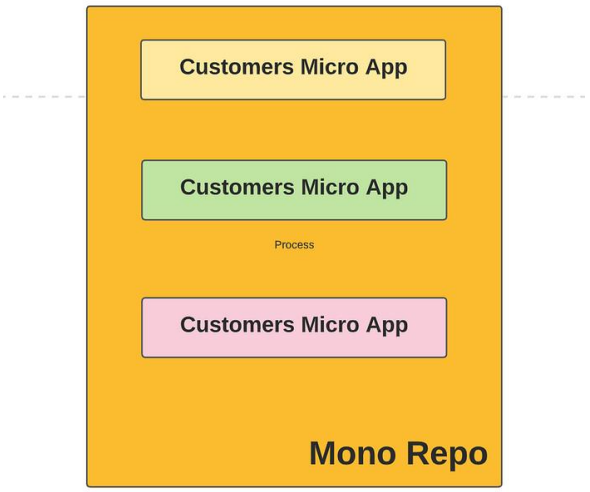
- Example Vue file (https: //gist. github. com/bbachi/3ce447fd98cd99ed0a385ad5ebb84801#file - example - vue)

In this architectural framework, the issue of communication between different components is effectively mitigated because all the individual libraries, once transformed into micro - frontends, are integrated into the same overarching project or application. This consolidation ensures that despite being developed as separate entities, the micro - frontends can interact seamlessly within the unified environment of the main application. The shared context and common runtime allow for straightforward data exchange and coordination between the components, facilitating a cohesive and synchronized operation. This approach not only simplifies the communication strategy but also enhances the application's maintainability and scalability by leveraging the modular nature of micro - frontends while maintaining a unified communication channel.

### 6) Monorepos

The monorepo strategy in software development is an approach where multiple interconnected projects are housed within a single repository. This methodology streamlines the management of shared code, eliminating the need to distribute common functionalities across separate repositories as libraries or modules for subsequent inclusion.

As depicted in the diagram below, a monorepo encompasses all projects along with their shared code. This configuration simplifies the development process by negating the requirement to externalize reusable code into distinct libraries. Instead, both the individual projects and the code they share reside within the same repository, facilitating easier access, modification, and maintenance of shared resources. This co - location of projects and shared code in a monorepo enhances collaboration among teams, ensures consistency across projects, and streamlines the build and deployment processes, thereby offering a cohesive and efficient framework for managing large - scale software developments.



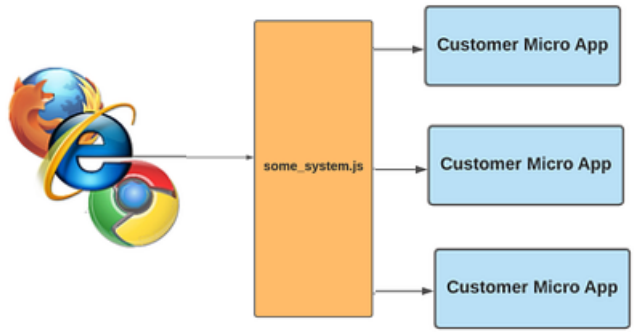

**Micro Frontends with Monorepos**

In a monorepo setup, a notable challenge arises in that every developer is required to clone the entire repository, even if their work only pertains to a few directories or components within it. This comprehensive checkout can lead to inefficiencies, particularly for those who only need access to a small segment of the repository for their tasks. Additionally, configuring build and deployment pipelines presents complexities under this model, given that all projects reside within a single repository. The intertwined nature of multiple projects in one repository complicates the task of setting up distinct pipelines tailored to each project's specific build and deployment requirements. Developers must meticulously define and manage these pipelines to ensure they accurately reflect the needs of each project while navigating the shared environment of the monorepo. This intricacy underscores the necessity for careful planning and execution in the adoption and management of a monorepo strategy, balancing its benefits with the operational challenges it introduces.

### 7) Customized Orchestrator

In this architectural approach, plain JavaScript files are utilized as orchestrators to manage the workflow of various micro - projects within the ecosystem. These orchestrator scripts are responsible for coordinating the overall operation, ensuring that each micro - project functions as part of a cohesive whole, despite being deployed independently. This independence of deployment allows for flexibility and scalability, as updates and modifications can be made to individual components without affecting the entirety of the system.

The orchestrator's role extends to dynamically loading each project in response to specific URL requests. This means that the orchestrator determines which micro - project to serve based on the user's navigation, seamlessly integrating disparate projects into a unified user experience. This mechanism not only enhances the responsiveness and efficiency of the application but also allows for a modular structure where components can be developed, tested, and deployed in isolation, yet operate harmoniously within the larger application framework.




**Micro Frontends with Customized Orchestrator**

In the orchestrator, identified in this context as some_system. js, it's possible to establish a global namespace along with specific objects dedicated to facilitating communication across the entire application ecosystem. This setup creates a universally accessible communication layer, whereby these globally defined objects become the medium through which data is transmitted and shared among the various projects within the system. This approach effectively allows for the creation of a centralized communication hub, enabling disparate projects to interact with one another by sending and receiving data through the global objects. Such a mechanism is crucial for ensuring that, despite the physical separation and independent operation of individual projects, there remains a cohesive and unified method for data exchange and interaction across the application. This strategy not only simplifies the process of inter - application communication but also promotes a modular architecture where components can remain loosely coupled yet functionally integrated.

## 2. Summary

- **Use of Iframes:** Embeds separate applications within a parent document, allowing for isolation but causing refresh with each application switch.
- **Webpack Module Federation:** Shares dependencies dynamically among various builds at runtime, enabling applications to be both independent and interconnected.
- **Monorepo Strategy:** Houses all projects and shared code in a single repository, simplifying code sharing but complicating pipeline configuration and requiring comprehensive checkout by developers.
- **Orchestration with Plain JavaScript:** Employs JavaScript files to manage and coordinate the workflow of micro - projects, with each project deployed independently and loaded dynamically based on URL navigation.
- **Global Namespace for Communication:** Establishes a global namespace in the orchestrator for inter - application communication, allowing data to be exchanged seamlessly among all projects.
- **Utilization of NGINX for Routing:** Leverages NGINX as a web server or reverse proxy to serve static content and route to appropriate micro - applications based on the context path, facilitating modularization.
- **Vue Libraries as Node Modules:** Packages each micro - frontend as a Vue. js library into a node module, enabling easy sharing and reuse across projects through a private repository.
- **Dynamic Imports and Lazy Loading:** Enhances performance by dynamically importing or lazily loading micro - frontends as needed, reducing initial load times and resource consumption.

## 3. Conclusion

In conclusion, the exploration of various approaches to implementing micro frontends reveals a diverse landscape of strategies, each with its unique strengths and challenges. From the isolation and integration capabilities of using Iframes and Webpack Module Federation to the centralized management and code sharing offered by the Monorepo Strategy, developers are equipped with a toolkit for crafting scalable and maintainable web applications. The orchestration of micro - projects through plain JavaScript and the facilitation of seamless inter - application communication via a global namespace underscore the importance of efficient coordination and data exchange. Meanwhile, leveraging NGINX for routing and packaging Vue libraries as node modules demonstrates the flexibility and modularity achievable in micro frontend architectures. The adoption of dynamic imports and lazy loading further enhances application performance, showcasing the potential for optimizing load times and resource utilization. Collectively, these approaches embody the evolving paradigm of web development, highlighting the potential for creating robust, modular, and user - centric web applications through the strategic implementation of micro frontends.

## References

[1] JavaScript Documentation https: //developer. mozilla. org/en - US/docs/Learn/Getting_started_with_the_web/JavaScript_basics
[2] Module Federation https: //webpack. js. org/concepts/module - federation/
[3] Iframes https: //developer. mozilla. org/en - US/docs/Web/HTML/Element/iframe
[4] VueJS Documentation https: //vuejs. org/guide/introduction. html
[5] Web Components https: //developer. mozilla. org/en - US/docs/Web/API/Web_components