Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2020, 7(10):61-65

Research Article ISSN: 2394 - 658X

Exploring Diverse Approaches to Micro-Frontends Integration with React

Bhargav Bachina

ABSTRACT

This paper introduces the concept of Micro-Frontends as a methodology to address the challenges of maintaining and collaborating on large-scale projects. As projects expand in size and complexity, Micro-Frontends offers a strategy to break down monolithic applications into smaller, more manageable components. By dividing projects into smaller teams and components, organizations can improve team performance and accelerate product delivery. The paper outlines six distinct approaches to implementing Micro-Frontends, providing a glimpse into each method's high-level architecture. Future discussions will explore these approaches in greater detail, offering practical guidance for their implementation in diverse technological environments.

Key words: Web Development, Programming, Software Development, JavaScript, React

---

INTRODUCTION

I have been working and researching Micro-Frontends for a while now and I have enough information to share it with the world. As projects get bigger and bigger it is very difficult to maintain and collaborate with these projects. As projects get bigger, their build time increases, unit tests increase, team size increases, almost everything gets bigger and gets to the stage where we can't maintain them. So, it's always better to have smaller projects and smaller teams for the teams' performance and deliver the products faster to the end-user. But sometimes your product or your app has so many features that maintaining smaller teams and projects is out of your hands. You can have separate teams for each feature but managing teams, merging all features into one repo, and resolving conflicts, etc. all these are tedious tasks that we can avoid with Micro Frontends.

Micro-Frontends are not a framework or library. This is the methodology where we can divide our fat apps into smaller and maintainable apps and design some kind of orchestration to place these apps in the browser window so that end-users see them as a single app. There are six different approaches that we could implement. Here are the 6 ways to do this irrespective of technology. I am going to do a separate post for each of these with full implementation. We will go through just the high-level architecture of these for now.

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

WEBPACK MODULE FEDERATION

Multiple separate builds should form a single application. These separate builds should not have dependencies between each other, so they can be developed and deployed individually.

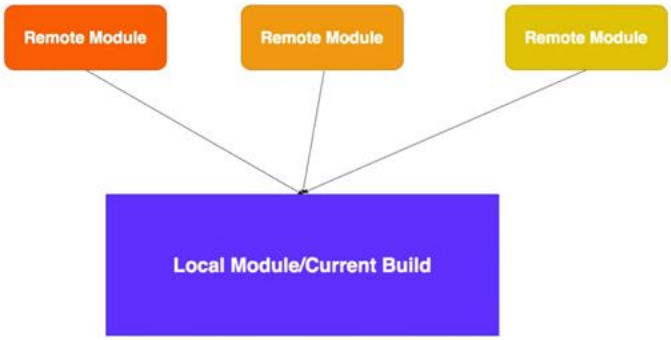

Figure 1: Webpack Module Federation

There are local and remote modules. Local modules are nothing, but the current build modules and remote builds are the builds that should be imported into the current App or build.

Each build acts as a container and consumes other builds as containers. This way each build can access any other exposed module by loading it from its container. It is possible to nest a container. Containers can use modules from other containers. Circular dependencies between containers are also possible.

You can find more here:

https://webpack.js.org/concepts/module-federation/

IFRAMES

Iframes are the HTML documents that can be embedded inside another HTML document. Here is an example of Iframe. You can place whatever source you want in the iframe tag, and it is rendered based on the width and height of the frame in the parent document.

https://gist.github.com/bbachi/4d45fb94ad2569b1d5583567350ab0b4#file-iframe-html

In the above example, I am running two React applications on ports **3000** and **3001** respectively and are the sources for the iframes defined in the above document. When you load this in the browser you can see two side by side in the same window as below. This is a simple example, but I am going to post a better example than this in the upcoming posts.

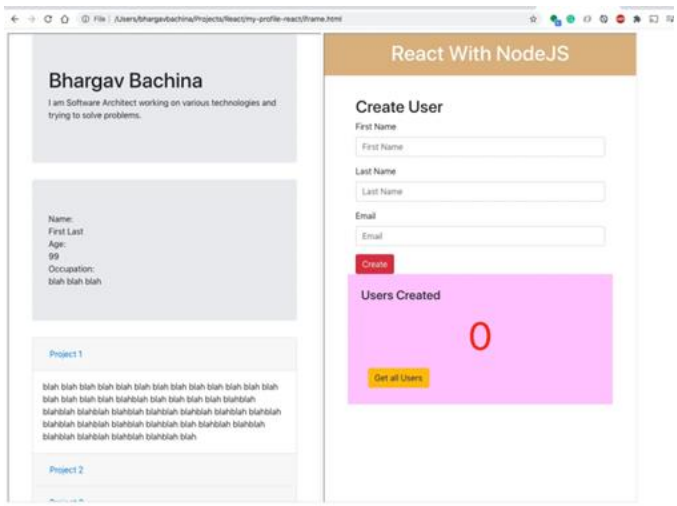

Figure 2: Micro Frontends with Iframes

This kind of approach better suits the project where all the functionality resides on the same page without any navigation and the communication happens through the Window object.

THROUGH NGINX

The above approach doesn't suit when there are navigation and routing involved in the project. You can have this but there should be another shell project for navigation. NGINX can be used as a web server or reverse proxy to serve static content. We can use NGINX to route the appropriate Micro App based on the context path. If we look at the following diagram, we have a NGINX web server in between to serve each Micro Frontend based on the context path or routing, for example, /**users** load the Micro Users, /**customers** loads the Micro Customers, and so on.

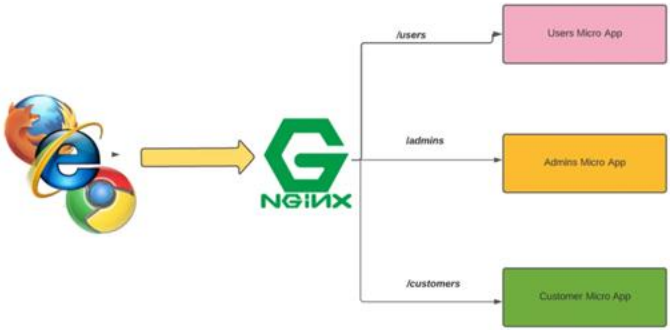

Figure 3: NGINX routing different apps based on context path

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.
https://gist.github.com/bbachi/698a8512667b3d3fa6ec2f0034e8f14b#file-nginx-conf
This kind of approach better suits the project where there is navigation or routing involved and when the app is divided into multiple apps based on the features. This communication happens through the React-Redux store and local storage as shown in the following diagram. As you navigate from one app to another app all the state of the app is transferred through some kind of state management tool and local storage.

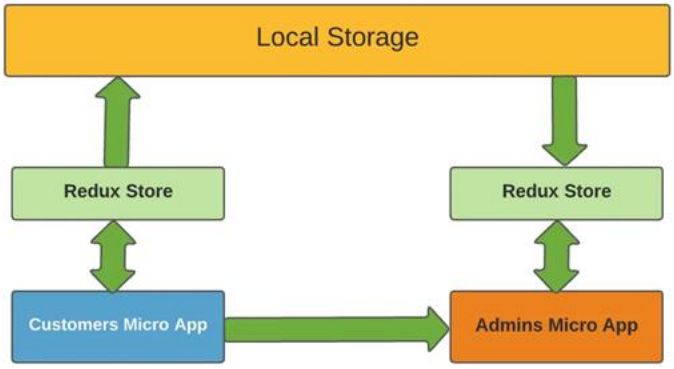

Figure 4: Communication between Micro Frontends

The disadvantage of this approach is that when we switch apps every time there is a page refresh. You might maintain some shared components on each app such as header and footer to maintain the same layout.

WEB COMPONENTS

Web components are the combination of different technologies that allow you to create reusable components on the web. It promotes the DRY principle. It mainly consists of three technologies **Custom elements** which allow you to create custom elements which can be used where you want on the web page, **Shadow DOM** which allows you to run your code in a separate DOM other than the main DOM that provides encapsulation, HTML templates which allow you write markup templates that can be used multiple times. You can get more information on this link: https://developer.mozilla.org/en-US/docs/Web/API/Web_components.

Figure 5: 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.

REACT COMPONENT LIBRARIES

In this approach, each Micro-Frontend can be a React library that can be pushed as a node module into some private repository. We have a shell app to pull that repo wherever we need with lazy loading (if it's Angular) or dynamic imports.
If you look at the below diagram, there are three Micro applications which can be converted to three libraries and pushed into the repository.

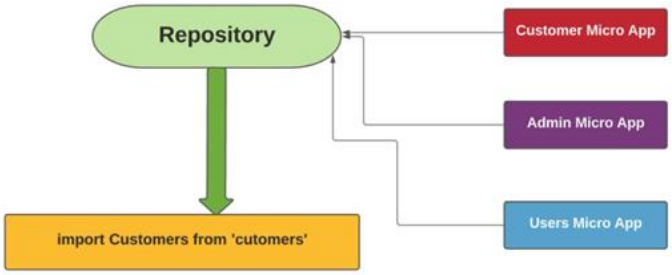

Figure 6: Micro Frontends with React Libraries

We can push our project as a node module with this command `npm push app1`We can import these libraries with dynamic imports as below.
https://medium.com/bb-tutorials-and-thoughts/7-different-ways-to-implement-micro-frontends-with-react-907b5e262230
Communication is not a problem here since all the individual libraries end up in the same project or application.

MONOREPOS

Monorepo is a software strategy where you can put multiple related projects under one repo. In this way, we don't have to push shared code to separate repo as a library or module and pull it for use.
If you look at the below diagram, we have one mono repo that contains all the projects and shared code as well. In this way, we don't have to create any separate libraries for the sharable code. All the shared code and actual projects live in the same repo.

Figure 7: Micro Frontends with Monorepos

Every developer has to check out the whole repo even he just needs one or two folders. Defining pipelines can be difficult with this approach as all the projects live in the same repo.

CUSTOMIZED ORCHESTRATOR

We define plain JavaScript files to orchestrate the entire workflow of these micro-projects. All individual projects are deployed independently. The orchestrator can load each project based on the URL.

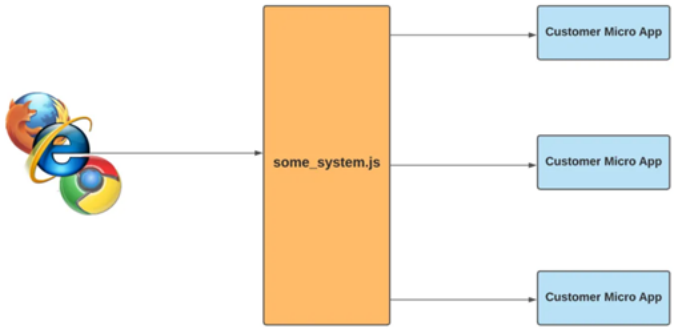

Figure 8: Micro Frontends with Customized Orchestrator

You can define some global namespace and objects in the orchestrator (some_system.js here) for communication. These global objects are available for all the projects so that you can send data among applications.

CONCLUSION

This paper presents Micro-Frontends as an effective strategy for mitigating the complexities associated with managing and collaborating on extensive projects. As project scale increases, Micro-Frontends provide a means to decompose monolithic applications into smaller, more modular elements. By fostering smaller teams and components, organizations can enhance team productivity and expedite product deployment. The paper delineates six unique methodologies for deploying Micro-Frontends, offering an overview of their architectural principles. Future explorations will delve deeper into these methodologies, furnishing actionable insights for their application across various technological landscapes.

REFERENCES

[1]. React Official Documentation https://react.dev/
[2]. Micro Frontend Article https://micro-frontends.org/
[3]. Official JavaScript Documentation https://developer.mozilla.org/en-US/docs/Web/JavaScript