VIEW POINT

APPLICATION DESIGN USING MICRO FRONTEND

Infosys®
Navigate your next

1. Introduction

In recent years, microservices have grabbed attention of organizations by overcoming limitations of large, monolithic backends. Businesses with large web apps which are continuously growing, it is difficult to maintain them in traditional monolithic frontend style. With their monolith codebase it's not easy to integrate new features, flexibility to implement a feature in the technology of choice and scaling separate component as they are tightly coupled.

As organizations have solved monolithic issues in backend, they are in a need of having the architectural pattern to avoid struggles of monolithic frontend codebases. The architectural design which avoids the issues of monolith frontend is **Micro frontend.**

Micro frontend Design

Micro frontend design is inspired from microservices which allows loose coupling of components. This design decomposes the large complex web application to smaller fragments which can be developed, tested, and deployed independently but still can be served as single product. Within this design, teams are vertically grouped to serve a particular feature.

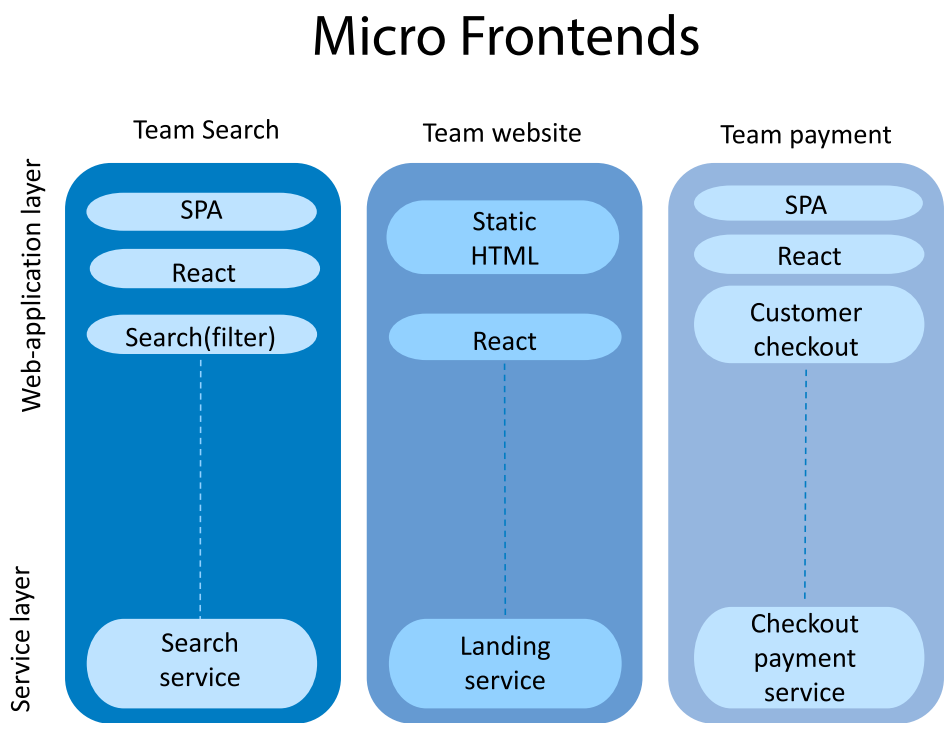


Fig. Micro frontends (source: medium)

Micro Frontend Application workflow

When a user visits a web page which is implemented in micro frontend design, the browser sends request to the index.html of container application then as per the browser request, container application renders the required micro frontend on the page.

The core ideas of Micro frontend design are,

- **Be technology independent:** Each team has the freedom to choose any frontend technology stack facilitates no coordination with other teams.
- **Team's code is isolated:** Teams which are using same framework will not share runtime facilitates no shared states or use of global variables in the application.
- **Robust web design:** Application features must be usable even though javascript is failed or yet to be executed. To improve performance methodologies such as "Universal Rendering" and "Progressive Enhancement" can be used.

Benefits of using micro frontend design

Better flexibility: Each micro frontend can be developed, tested, deployed independently. So, if team A is working on bug fix and team B wants to deploy their feature, team B does not need to wait for team A to finish their work.

Freedom to choose technology: UI teams developing different features can choose different technology stack as per business requirements.

Multiple, smaller codebases: Each UI fragment's team have their own codebases which are small, manageable, involves simple code reviews and only few developers will be a part of certain UI fragment.

Easy scaling of micro apps: In monolithic frontend approach, if a new feature is added, we have to scale the whole application. But in micro frontend approach, each micro app can be scaled independently which results in less time to market than monolithic frontend.

Autonomous teams and systems: Each team can work on their feature with minimal dependency on other teams so that if other components are down still it can function.

Easy to get started: Micro frontend is easy to understand and manage than monolithic frontend thereby new developers don't need to spend a lot of time to understand the codebase before adding value to the project.

Business use cases where we can think of micro frontend as a good idea

- Scenarios of having more developers working for a large business domain, planning to reduce complexity by splitting the whole application into multiple subdomains and want to deploy each feature of the entire application independently with no overloaded coordination and communication between subdomains.
- Use cases like building innovative, responsive, and scalable digital applications.
- To improve overall productivity or productivity focused project, micro frontend approach can be a good idea.

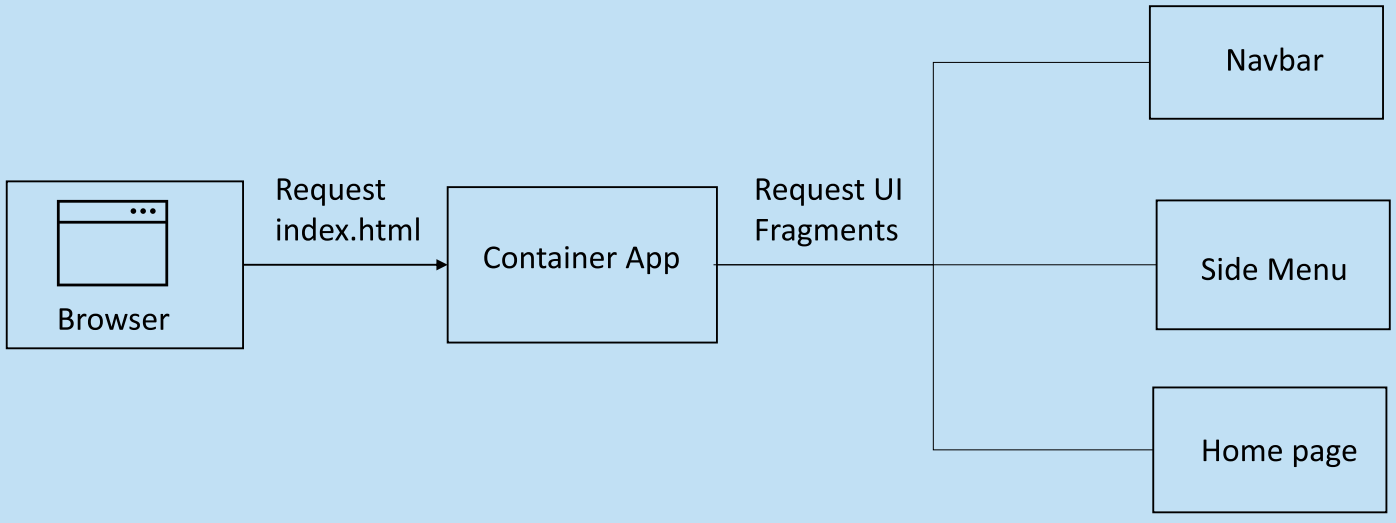


Fig. Micro frontend application workflow (source: medium)

2. Implementation Techniques

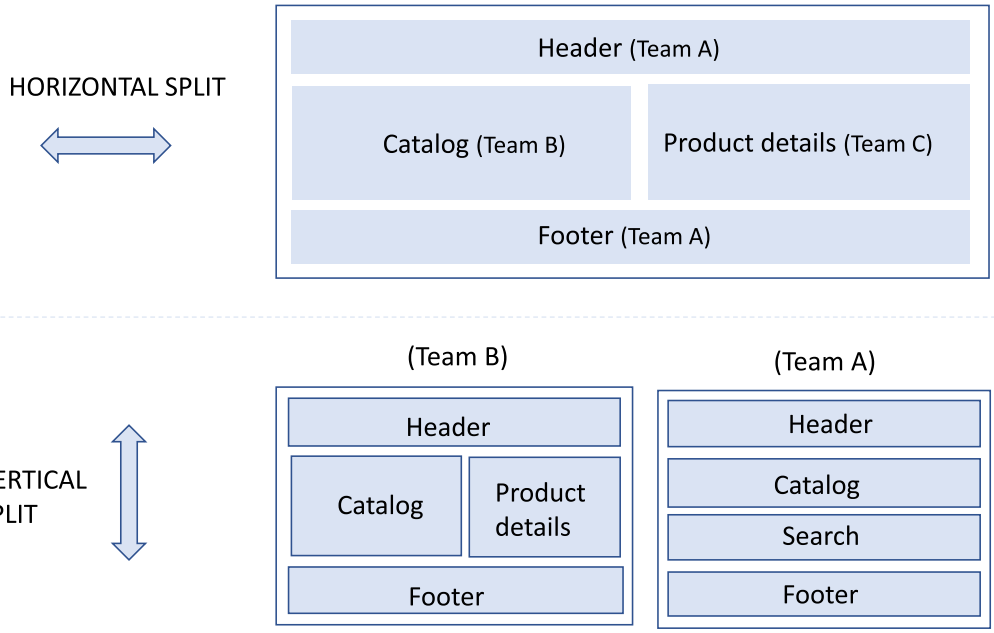
Before moving to implementation options of micro frontend, we have to decide how UI has to be split.

Basically, there are two ways to split application frontend which are,

Horizontal split: The horizontal split splits an interface into multiple parts, assign them to different teams.

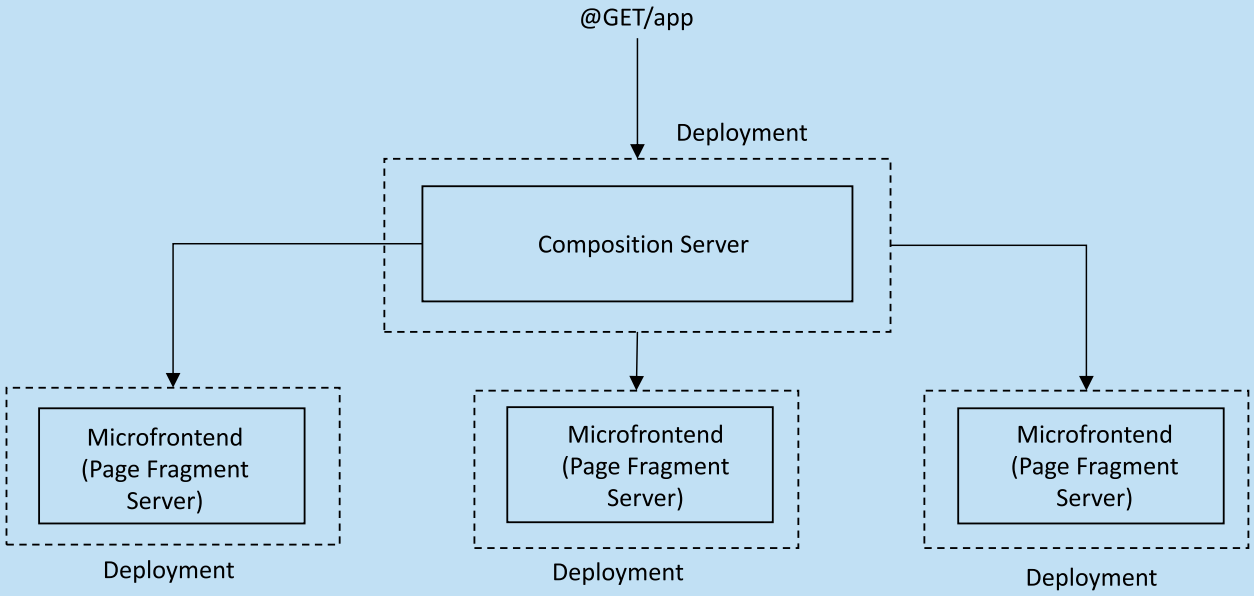
Vertical split: Here in vertical split, based on business domain priority, each domain will be assigned to different teams.

There are multiple ways to implement micro frontend design pattern based on where and how different micro frontends are composed.



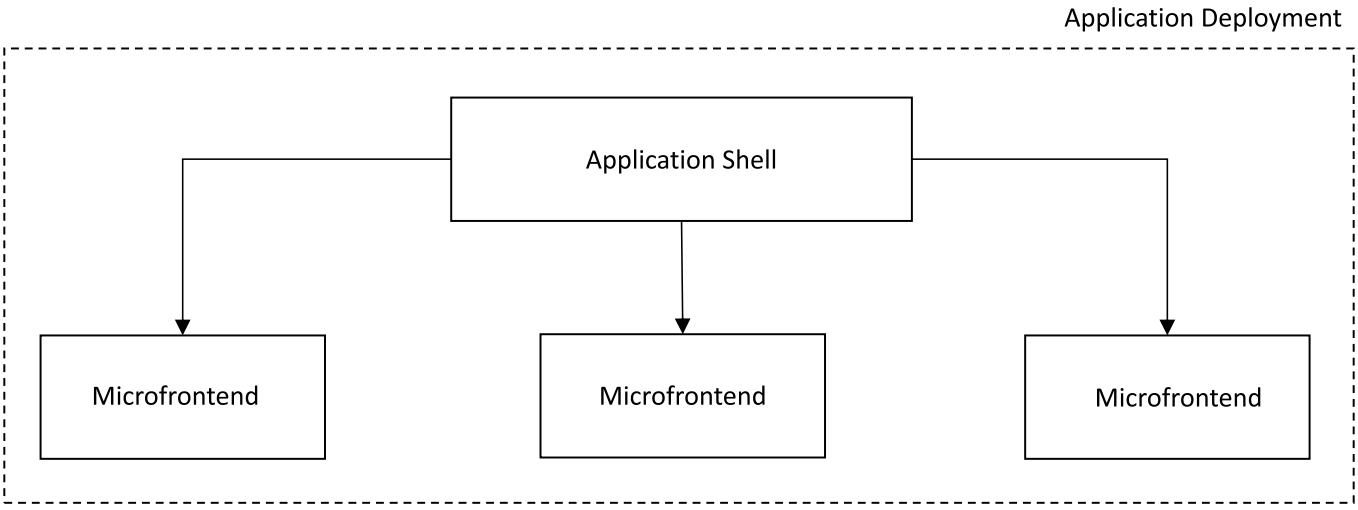
Server-side composition

In this approach, as the name says different views composition happens at server level. The server calls micro frontends, composes the view, and organizes different components of the page before it serves. And also we can always load the base component on server, avoids blank screen or unnecessary loading time to users. Eg. Nginx, Podium, Tailor.



Build-time integration

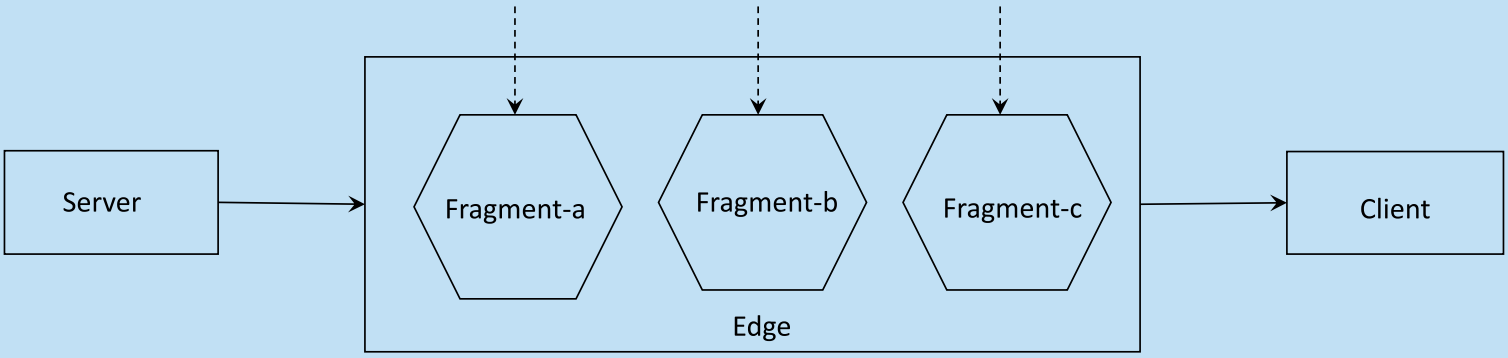
Composing components at build-time can be implemented by stating micro frontend components as npm dependencies in a package.json file. This approach facilitates easy deployment but the major drawback of this approach is that increased dependency between different micro frontends makes it difficult to have independent pipelines, results in same bottlenecks of monolithic frontend. Businesses still use this approach as per their use cases. Eg. Shared libraries, SSG Fragments, Bit.



Edge-Side Composition

In this approach, micro frontends are composed at edge layer, such as CDN.

The below given implementation techniques are client side integration, follow the fundamental principle of client side integration where different micro frontends are delivered to the browser. Once sourced, these micro frontends can be placed and organized in multiple ways as per the following:



Run-time via iframes

This integration technique serves principles of micro frontend design, easy to deploy and also enables isolation between parent application and micro frontends. But this technique limits UX experience to iframe boundaries also adds more complexity while building responsive page.

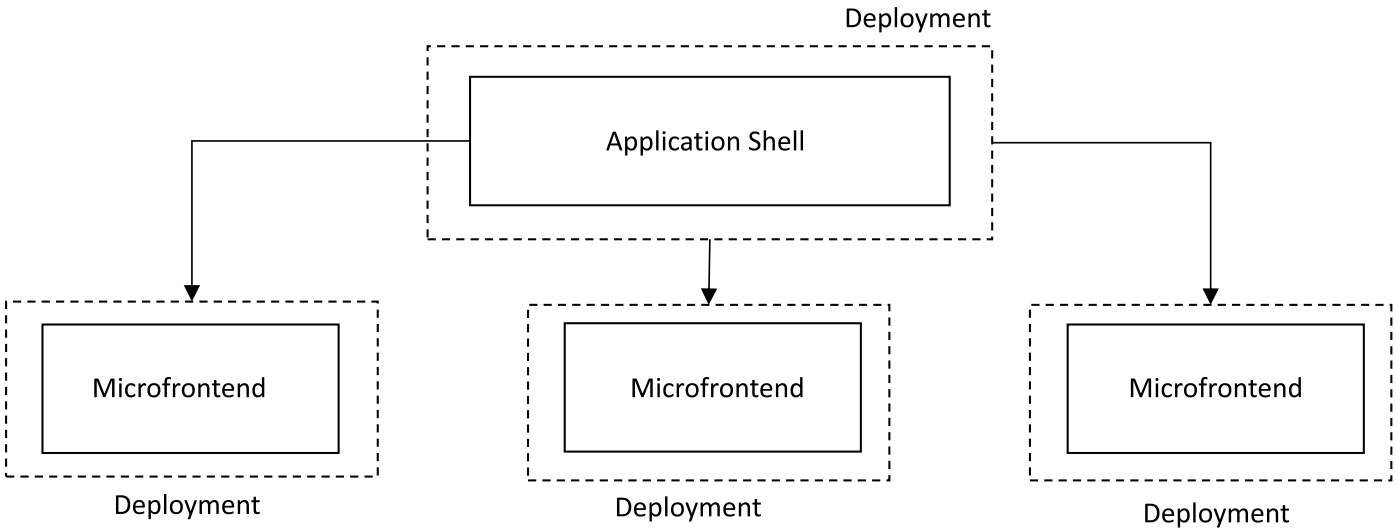
Run-time via JavaScript

JavaScript overcomes the limitations of iframes. With client side composition, we can easily find what are the required micro frontends and when and where it should be rendered.

Run-time via web components

This technique is slightly different from the above techniques. Here, Web components refer to micro frontends whereas in above techniques they are referred as bundles. So these web components can respond to URL routing and can render micro frontends as per other run time integration techniques.

There are many frameworks available to implement micro frontend. The most popular framework is **Webpack Module Federation Plugin** (Client-Side composition).



3. Module Federation based Integration

Various integration approaches are given to fulfill different requirements, but if we see from the benefits and flexibility perspective Module Federation based integration is widely accepted to share code and libraries between applications.

With module federation, any JavaScript code, such as business logic or libraries, and state management code can be shared between applications. It can make multiple micro frontend apps work together as if you develop a monolith app. The purpose of module federation is to share JavaScript code between applications, not UI. So, the shared code can be used in an app.

With module federation we can avoid code duplication and provide shared UI components across the entire app.

4. Deep Dive into Module Federation based Integration

Before starting to integrate the projects, our apps (host app as well as remote apps) require some setup to be aware about other projects.

We need to create a "webpack.config.js" file at the root of **host/** :

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin =
require("webpack/lib/container/ModuleFederationPlugin");
const { dependencies } = require("./package.json");

module.exports = {
  entry: "./src/index",
  mode: "development",
  devServer: {
    port: 3000,
  },
  module: {…},
  plugins: [
    new ModuleFederationPlugin({
      name: "Host",
      remotes: {
        Remote1: `Remote1@http://localhost:4000/appEntry.js`,
        Remote2: `Remote2@http://localhost:5000/appEntry.js`,
      },
      shared: {
        ...dependencies,
        react: {
          singleton: true,
          requiredVersion: dependencies["react"],
        },
        "react-dom": {
          singleton: true,
          requiredVersion: dependencies["react-dom"],
        },
      },
    }),
    new HtmlWebpackPlugin({
      template: "./public/index.html",
    }),
  ],
  resolve: {…}
```

- **entry:** this defines which file to target when project is started
- **mode:** this decides which environment file to load.
- **devServer:** this contains the port number for the project
- **ModuleFederationPlugin:** this allows us to combine the applications at runtime in the client's browsers.
  - **name:** it is used to distinguish between the modules. It is not important in this example as we are not exposing this anything in this project, but it is necessary for our Remote projects.
  - **remotes:** it is where we define the federated modules we want to consume in this app. You'll notice we specify Remote as the internal name so we can load the components using import("Remote/<component>"). But we also define the location where the remote's module definition is hosted: "Remote@http://localhost:4000/appEntry.js". This URL tells us three important things. The module's name is Remote, it is hosted on localhost:4000, and its module definition is appEntry.js.
  - **shared:** this is how we share dependencies between modules. This is very important for React because it has a global state, meaning you should only ever run one instance of React and ReactDOM in any given app. To achieve this in our architecture, we are telling webpack to treat React and ReactDOM as singletons, so the first version loaded from any modules is used for the entire app, if it satisfies the requiredVersion we define. To minimize the number of duplicate dependencies between our modules we are also importing all our dependencies from package.json and include them here.

Now we will start configuring our remote apps, for that we will create webpack.config.js file in the root of **remote1**/ and **remote2/:**

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const ModuleFederationPlugin =
require("webpack/lib/container/ModuleFederationPlugin");
const path = require("path");
const { dependencies } = require("./package.json");

module.exports = {
  entry: "./src/index",
  mode: "development",
  devServer: {
    static: {
      directory: path.join(__dirname, "public"),
    },
    port: 4000,
  },
  module: {…},
  plugins: [
    new ModuleFederationPlugin({
      name: "Remote1",
      filename: "appEntry.js",
      exposes: {
```

```
        "./App": "./src/App",
      },
      shared: {
        ...dependencies,
        react: {
          singleton: true,
          requiredVersion: dependencies["react"],
        },
        "react-dom": {
          singleton: true,
          requiredVersion: dependencies["react-dom"],
        },
      },
    }),

  ],
  resolve: {…},
};
```

- Our webpack dev server runs at localhost: 4000
- The remote module's name is Remote
- The filename is moduleEntry.js

Combining these will allow our host to find the remote code at "Remote1@http://localhost:4000/appEntry.js"

- **Exposes:** it is where we define the code we want to share in the appEntry.js file. Here we are exposing: <App />.

*We will do the same thing for Remote2

Now we are done with the configuration, we can start our projects by performing npm start inside each individual project directories, i.e., host, remote1 and remote2, all projects will start working on their assigned ports.

Now to use remote project inside the component of our host project, we will import the remote project in the file with:

```
const RemoteApp1 = React.lazy(() => import("Remote1/App"));
```

and use with:

```
<RemoteApp />
```

For handling navigation based on any event from remote app, we have created a event listener inside in our host app and navigate according to the data passed:

```
useEffect(() => {
    const portalNavigationEventHandler = (event) => {
      const eventDetail = event.detail;
      if (location.pathname === eventDetail.pathname) { return;}

      if (eventDetail.pathname.startsWith(APP_BASE_URL)) {
          navigate({ pathname: eventDetail.pathname});
      }
    };
    window.addEventListener(
      "[RemoteApp] navigated",
      portalNavigationEventHandler
    );

    return () => {
      window.removeEventListener(
        "[RemoteApp] navigated",
        portalNavigationEventHandler
      );
    };
  }, [location]);
```

The useEffect Hook allows you to perform side effects in your components. Some examples of side effects are: fetching data, directly updating the DOM, and timers. useEffect accepts two arguments. The second argument is optional. useEffect(<function>, <dependency>).

5. Challenges Faced

Faced challenges in integrating search auto complete and navigation triggered from remote apps.

We resolved search auto complete by using the package 'react-search-autocomplete', but the package was firing search function instantly as the user typed so to overcome this we created custom input debounce function so that before calling the search function it will wait for given amount of time.

And, to resolve the navigation triggered from remote apps we used the javascript based event bus which will listen to certain event and will perform the action accordingly.

6. Conclusion

Business use cases where micro frontends are the right choice:

- Scenarios of having more developers working for a large business domain, planning to reduce complexity by splitting the whole application into multiple subdomains and want to deploy each feature of the entire application independently with no overloaded coordination and communication between subdomains.
- Use cases like building innovative, responsive, and scalable digital applications.
- To improve overall productivity or productivity focused project, micro frontend approach can be a good idea

Authors

Aman Srivastava
Specialist Programmer

Vellimani M
Digital Specialist Engineer

Mentor

Sachin Girijashankar Agrawal
Lead Architect

8. References

- https://martinfowler.com/articles/micro-frontends.html
- https://medium.com/frontend-at-scale/an-introduction-to-micro-frontends-1a43edb4c38e
- https://dev.to/okmttdhr/micro-frontends-patterns-11-23h0

For more information, contact askus@infosys.com

Infosys®
Navigate your next