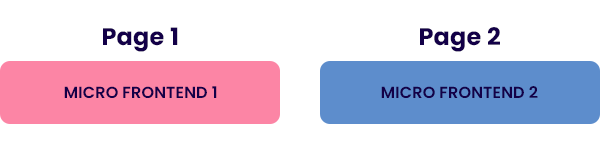
**Micro Frontend Approaches to Split the page:**

* vertical split



* horizontal split

A screenshot of a website

Description automatically generated

**Micro Frontends Composition (Types of Rendering)**

* Client-Side - Each microfrontend is assembled by browser server
* Edge-side (Different from pre-rendering) - ESI is a markup language that allows you to assemble web pages from smaller fragments, which can be cached and assembled at the edge (CDN)
* Server-side - Each microfrontend is assembled as SSR file and server to browser by server

**Types of Approach**

* Traditional way - Hosting and Importing Build Output (e.g., Bundle.js)
* Remote Containers or Remote Entry Points (eg: Module federation and Single spa)
* IFrame Embedding – less common(not covered)

**Traditional way**

***Working:***

Build and bundle the application into single js file, host the bundled file and import it as a script in some other application.

***Steps to achieve it:***

Host the bundle of MicroFrontend application:

* Create a simple react application and install webpack related dependencies in it.
* Create a separate Class or Functional component for creating a custom tag by attaching our actual component (App) into shadow DOM.

Eg:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

class ExternalUserApp extends HTMLElement {

constructor() {

super();

this.attachShadow({ mode: 'open' }); // Create a shadow DOM

}

// Overridden method of HTMLElement

connectedCallback() {

const shadow = this.shadowRoot;

const root = document.createElement('div');

shadow.appendChild(root);

// Render your React application content here

ReactDOM.render(<App />, root); // Replace with your actual component

}

}

customElements.define('external-users-app', ExternalUserApp);

export default ExternalUserApp;

Above code allows you to create a reusable custom element('external-users-app') that encapsulates a React application within its shadow DOM. You can then use this element in your HTML(with help of HTMLElement) like any other HTML tag

* Configure webpack to build the application into a single js output file (entry point should be above created Class or functional component)

Eg:

//webpack.config.js

const path = require('path');

module.exports = {

entry: './src/ExternalUserApp.js',

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist'),

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader',

options: {

presets: ['@babel/preset-env', '@babel/preset-react'],

},

},

},

],

}

};

* Build the application using ***npx webpack --config webpack.config.js --mode production*** (bundle.js will be created in dist folder as configured in webpack.config.js)
* Now you can host the created build (CDN for production). In local you can host with help of http-server(Should be added as dependency)

Cmd from root path: ***http-server dist -p 3001***(verify it by hitting <http://localhost:3001/bundle.js>)

Use MicroFrontEnd Application bundle in Parent application

* Create a new simple parent react application to import above hosted bundle as Script
* Import the script in head or body part of public/index.html in newly created application

<script src="http://localhost:3001/bundle.js"></script>

* Use the custom tag which we have created in our microfrontend (step 2) which will be present in imported bundle.js as script, on our newly created application

function App() {

return (

<div>

<external-users-app></external-users-app>

</div>

);

}

export default App;

Like this you can created multiple custom tags on multiple application and use it in a application as MicroFrontend

**Remote Containers**

Modules are hosted in separate remote servers which can dynamically composed, and dependencies can be shared

**1) Module federation:**

Module gets exposed by a webpack plugin of one micro frontend which can used as remote in root micro frontend

* Create simple react app and Install webpack related dependency *"webpack webpack-cli webpack-dev-server babel-loader css-loader html-webpack-plugin http-server" (update build, start cmd as well)*
* Create an entry file in src and configure it to import index.js

// src/entry.js

import('./index.js')

* Now entry.js is imported in index.js(whole app start point) we can give it as entry point for creating webpack build in webpack.config.js

// webpack.config.js

const HtmlWebpackPlugin = require("html-webpack-plugin");

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

const { dependencies } = require("./package.json");

module.exports = {

entry: "./src/entry.js",

mode: "development",

devServer: {

port: 3003,

},

module: {

rules: [

{

test: /\.(js|jsx)?$/,

exclude: /node\_modules/,

use: [

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/i,

use: ["style-loader", "css-loader"],

},

],

},

plugins: [

new HtmlWebpackPlugin({

template: "./public/index.html",

favicon: "./public/favicon.ico",

}),

new ModuleFederationPlugin({

name: "ExternalUserApp",

filename: "remoteEntry.js",

exposes: {

"./ExternalUser": "./src/App",

},

shared: {

...dependencies,

react: {

singleton: true,

requiredVersion: dependencies["react"],

},

"react-dom": {

singleton: true,

requiredVersion: dependencies["react-dom"],

},

},

}),

],

resolve: {

extensions: [".js", ".jsx"],

},

target: "web",

};

* Now to expose a specific module from build you can use ModuleFederation plugin like above.
* Once you do webpack build, remoteEntry.js will be created as configured above.. which contains the module to be exposed.
* Start the application (webpack serve) and check in browser (<http://localhost:3003/remoteEntry.js>) for exposed module build.

Using the exposed build in Root application

* Create simple react app for root and Install webpack related dependency *"webpack webpack-cli webpack-dev-server babel-loader css-loader html-webpack-plugin http-server" (update build, start cmd as well)*
* Create webpack.config.js and configure the remote to use above hosted module into root application

const HtmlWebpackPlugin = require("html-webpack-plugin");

const ModuleFederationPlugin = require("webpack/lib/container/ModuleFederationPlugin");

const { dependencies } = require("./package.json");

module.exports = {

entry: "./src/entry",

mode: "development",

devServer: {

port: 3005,

},

module: {

rules: [

{

test: /\.(js|jsx)?$/,

exclude: /node\_modules/,

use: [

{

loader: "babel-loader",

options: {

presets: ["@babel/preset-env", "@babel/preset-react"],

},

},

],

},

{

test: /\.css$/i,

use: ["style-loader", "css-loader"],

},

],

},

plugins: [

new HtmlWebpackPlugin({

template: "./public/index.html",

}),

new ModuleFederationPlugin({

name: "RootApp",

remotes: {

"ExternalMf": "ExternalUserApp@http://localhost:3003/remoteEntry.js",

},

shared: {

...dependencies,

react: {

singleton: true,

requiredVersion: dependencies["react"],

},

"react-dom": {

singleton: true,

requiredVersion: dependencies["react-dom"],

},

},

}),

],

resolve: {

extensions: [".js", ".jsx"],

},

target: "web",

};

* Now you can lazy load your exposed module in root app

const ExternalUserApp = lazy(() => import('ExternalMf/ExternalUser'));

function ExternalUserComponent() {

return (

<div>

<Suspense fallback={<div>Loading Header...</div>}>

<ExternalUserApp />

</Suspense>

</div>

);

}

export default ExternalUserComponent;

* Configure the above component in App.js and Start the application (webpack serve) and check in browser (<http://localhost:3005>) to see microfrontend content in root app.

***Extra information about webpack:***

When you use webpack for build and serve. You need to specify below content in public/index.html

<script src="./dist/bundle.js"></script>

If you start in locally above code is not necessary. But when deploying in server this is necessary because server need to know the path of bundle.js

To avoid it doing manually we can add a below plugin in webpack.config.js.

new HtmlWebpackPlugin({

template: "./public/index.html",

favicon: "./public/favicon.ico",

}),

This will create a index.html inside dist (build folder) with the above script tag added automatically. This index.html is served automatically to browser.

**2) Single SPA:**

It is a framework which is specifically used to create microfrontend with different framework or libraries(Angular, Vue, react, others) because of ease of development and features it have.

Note:

Using Single Spa alone we cannot build a MicroFrontend application. It should be combined with either systemjs-importmap (recommended) or Module federation. Because it is necessary to expose a microfrontend to another microfrontend to integrate. So for that importmap or Module federation is used.

*Why I need to use single spa with module federation instead I can directly use module federation?*

Single SPA handles the overall orchestration, lifecycle and routing, while Module Federation takes care of shared dependencies, Shared State and dynamic module loading.

While Module federation just a plugin which cannot do all these.

You can create an application using Single SPA by two way:

* Normal creation of react app and configure Routing, lifecycle and other things or Using create-single-spa command(easy, recommended and used below) which make it more simple to create microfrontend application.
* Install single spa library

npm i -g create-single-spa

* Start creating application using singlespa command

npx create-single-spa

* Generally you need to create at least 2 application – 1 microfrontend application, 1 – root/base application which configures multiple microfrontend in it.
* Choose *single-spa application/parcel* for creating microforntend app and next time(after creating mf app) choose *single-spa root config* for creating root/base application to add microfrontend in it.

? Directory for new project test

? Select type to generate (Use arrow keys)

❯ single-spa application / parcel

in-browser utility module (styleguide, api cache, etc)

single-spa root config

* Choose framework/library(React, Vue, Angular) you want
* If you are choosing single-spa root config give ‘Y’ for single-spa Layout Engine

? Would you like to use single-spa Layout Engine (Y/n)

* Provide the organization/project name like your wish
* Now let’s assume we have created a micro frontend application and a root application.

Start microfrontend application

* Using npm start:standalone you can start the microfrontend application.
* Go to browser and check whether the application is up or not. Also you application is build will be hosted by systemjs on <host>:<port>/<org-name>-<project-name>.js. Copy that url.

A screenshot of a computer

Description automatically generated

Configure microfrontend application in your created root/base application

* Now you can add the copied URL in above step as importmap inside index.ejs file present inside root-app .

//src/Index.ejs

<% if (isLocal) { %>

<script type="systemjs-importmap">

{

"imports": {

"@external/external-user-app": "http://localhost:8080/external-external-user-app.js",

"@root/root-config": "http://localhost:9000/root-root-config.js"

}

}

</script>

<% } %>

"@external/external-user-app": <http://localhost:8080/external-external-user-app.js> - This can be any name (“ext” instead "@external/external-user-app”)

"@root/root-config": <http://localhost:9000/root-root-config.js> - This name for root/base app is predefined

@root/root-config (based on org name and project name given while creating). This should not be changed because this format is referred internally by systemjs. Consider 9000 is the port on root/base application is started.

* Now configure your microfrontend route (as per imported name) inside microfrontend-layout.html of root-app (file created by single spa framework)

<single-spa-router>

<main>

<route path="/external">

<application name="@external/external-user-app"></application>

</route>

</main>

</single-spa-router>

* Now start the application using npm run start
* You can see content of the microfrontend application inside root application on configured (‘/external’) path

A screenshot of a computer

Description automatically generated

**Data sharing between MicroFrontEnds**

Module federation - you use a global state library (like Redux) and share it across microfrontends, you can synchronize state updates between them.

SingleSpa with Systemjs – Not straight forward, using global event bus we can share data as events