

MicroFrontend for “Dummies”

Informe sobre implementación y creación de un
proyecto desde cero de MicroFrontend con
Single-SPA



TELECOM



Elaborado por:

Emilio Buchailot
Maximiliano de Torres
Leandro Avram
V1.0

Índice

Introducción	3
¿Quiénes somos?	3
Objetivos	3
Primeros pasos	3
Experiencias	4
Herramientas utilizadas	5
Single SPA	5
Frameworks de Javascript	5
Repositorio	5
Arquitectura a implementar	6
Capturas finales.	7
Aplicación principal	8
Crear main-app	8
Instalar single-spa	8
App.js	8
Carpeta apps	9
Utils	10
Index.js	10
Creación subaplicación Login	11
Crear la aplicación	11
Instalar single-spa para react	11
Ciclo de vida SingleSpaReact	11
Registrar subaplicación login	12
Agregar tags de subaplicación en aplicación principal	14
Llamar al registerLoginAppReact e iniciar single-spa	14
Utilización de webpack en login	15
Creación subaplicación navbar	17
Crear aplicación	17
Instalar single-spa-vue	17
Ciclo de vida SingleSpaVue	17
Registrar subaplicación navbar	19
Agregar tags subaplicación en aplicación principal	20
Llamar al registerNavbarAppVue	20
Webpack navbar	21
Mejoras pendientes	22
Creación subaplicación Grilla	22

Instalar single-spa para angular	22
Componente Grilla	22
Registro en Aplicación Principal	23
Principales inconvenientes single-spa-angular	25
WebComponent JS	27
Getters - Setters	27
Ciclo de Vida	28
Render	29
Declaración de custom element	30
Utilización	30
Comunicación y GlobalEventDistributor	32
Redux Main-App	32
Store y Constants	34
GlobalEventDistributor	35
Inyección de GlobalEventDistributor en aplicación de Login	35
Inyección de GlobalEventDistributor en aplicación de Navbar	37

Introducción

¿Quiénes somos?

El equipo de desarrollo de esta prueba de concepto basada en la experimentación de Microfrontend está conformado por estudiantes avanzados de la Universidad Tecnológica Nacional - Facultad Regional Córdoba siendo supervisado, apoyado y dirigido por Telecom-Fibertel Argentina.

Objetivos

El objetivo principal de este documento es demostrar y enseñar de manera sencilla paso a paso cómo desarrollar una pequeña aplicación utilizando Single-SPA y WebComponents, permitiendo de manera exitosa arrancar en la utilización de esta técnica/tecnología llamada MicroFrontend.

Primeros pasos

Nosotros contábamos con experiencia previa en el desarrollo de pruebas de concepto con los frameworks más utilizados de Javascript como lo son Vue, React y Angular, lo que resultó en que esta demostración y prueba de concepto no se hizo tanto hincapié en la implementación de cada uno de estas tecnologías y sus características, sino en cómo se puede generar sinergia entre los mismos para el desarrollo de diferentes funcionalidades de una aplicación mayor.

Así que teniendo conocimientos previos de Javascript, de Webpack y alguno de los frameworks que vamos a utilizar puede resultar de gran ayuda a la hora de comenzar con este desarrollo y nosotros quedamos atentos a recomendaciones u opiniones de la comunidad, ya que lejos de ser expertos en cada uno de las herramientas utilizadas se llegó a buen puerto.

Nuestros primeros pasos fueron guiados por varios tutoriales y ejemplos, principalmente agradecemos la ayuda y el código de ejemplo del grupo de desarrolladores Pragmatists de Polonia y su repositorio de Microfrontend con Single-SPA, también a Joel Denning por algunos de sus ejemplos en distintos framework y ser uno de los contribuyentes más activo del repositorio Canopytax/single-spa.

<https://github.com/Pragmatists/microfrontends>

<https://blog.pragmatists.com/independent-micro-frontends-with-single-spa-library-a829012dc5be>

<https://github.com/joeldenning>

<https://github.com/CanopyTax/single-spa>

Experiencias

Javascript, navegadores y en general cualquier tecnología de frontend evoluciona a un ritmo acelerado, lo cual hace que muchas funcionalidades y/o librerías no son del todo compatibles a medida que pasa el tiempo lo que hizo este camino siempre variable y con muchas re-implementaciones de lo que se creía ya implementado. Lo cual no es un panorama nada desalentador para nosotros porque esto se transforma en una oportunidad de estar constantemente actualizados a lo que ofrecen los frameworks, metaframeworks y librerías de Javascript y nunca perder de vista que actualizaciones van teniendo los navegadores, como así también qué funcionalidades quedan deprecadas en los mismos.

Un caso puntual de aprendizaje/actualización se nos presentó con la librería single-spa-angular, en nuestra primera prueba de concepto creamos una subaplicación en angular 5 con webpack personalizado para buildear y comenzar a usar la aplicación. Quisimos crear una nueva aplicación en angular pero en su última versión estable la 8.0.3 y nos encontramos con que la librería single-spa no tenía soporte para angular 8, a los pocos días (3 o 4) volvimos a ingresar a single-spa-angular y ya habían actualizado la librería a una versión release candidate, a los pocos días está versión ya se convirtió en un release estable que no solo daba soporte a angular 8, sino que también agregaron nuevas características, tal es así que genera un archivo main.single-spa.ts en la ruta ./src. con todo el ciclo de vida, genera archivo asset-url.ts en ./src/single-spa con un método para utilizar las imágenes que se guardan en asset, crea un archivo custom webpack en la que se encapsula su utilización y agrega un script npm run build:single-spa y serve: single-spa en package.json. Por lo que vimos que la librería está constantemente mejorando y actualizando las configuraciones relacionadas con la utilización de la librería dejando al desarrollador que se enfoque solo en la implementación de la funcionalidad de la aplicación. Las aplicaciones que fuimos creando para implementar MicroFrontend, utilizamos webpack para realizar la carga de los módulos de cada subaplicación en la aplicación principal, esto nos trajo un poquito de complejidad, ya que tuvimos que aprender a configurar cada webpack según el framework utilizado.

Como desafío nos queda realizar una prueba de concepto utilizando SystemJs en la aplicación principal para realizar la carga de los módulos de cada subaplicación en lugar de configurar cada uno de los webpack.

Herramientas utilizadas

Single SPA

Single-spa es un meta-framework para unir múltiples MicroFrontend de javascript en una aplicación de frontend.

La arquitectura del frontend utilizando single-spa brinda los siguientes beneficios:

- Usar múltiples framework(angular,react,ember,vue,etc) en la misma página, sin tener que actualizarla.
- Deployar los frontends de forma independiente.
- Código de carga diferida(lazy load) para mejorar el tiempo de carga inicial.

Single-spa funciona con ES5, ES6+, TypeScript, Webpack, SystemJs, Gulp, Grunt, Bower, ember-cli, o cualquier sistema de compilación disponible. Se puede utilizar npm install, jspm install, o usar los tags <script>. También funciona con los diferentes navegadores que se utilizan en la actualidad, Chrome, Firefox, Safari, IE11 y Edge.

Frameworks de Javascript

1. Angular:
2. ReactJS:
3. VueJS:

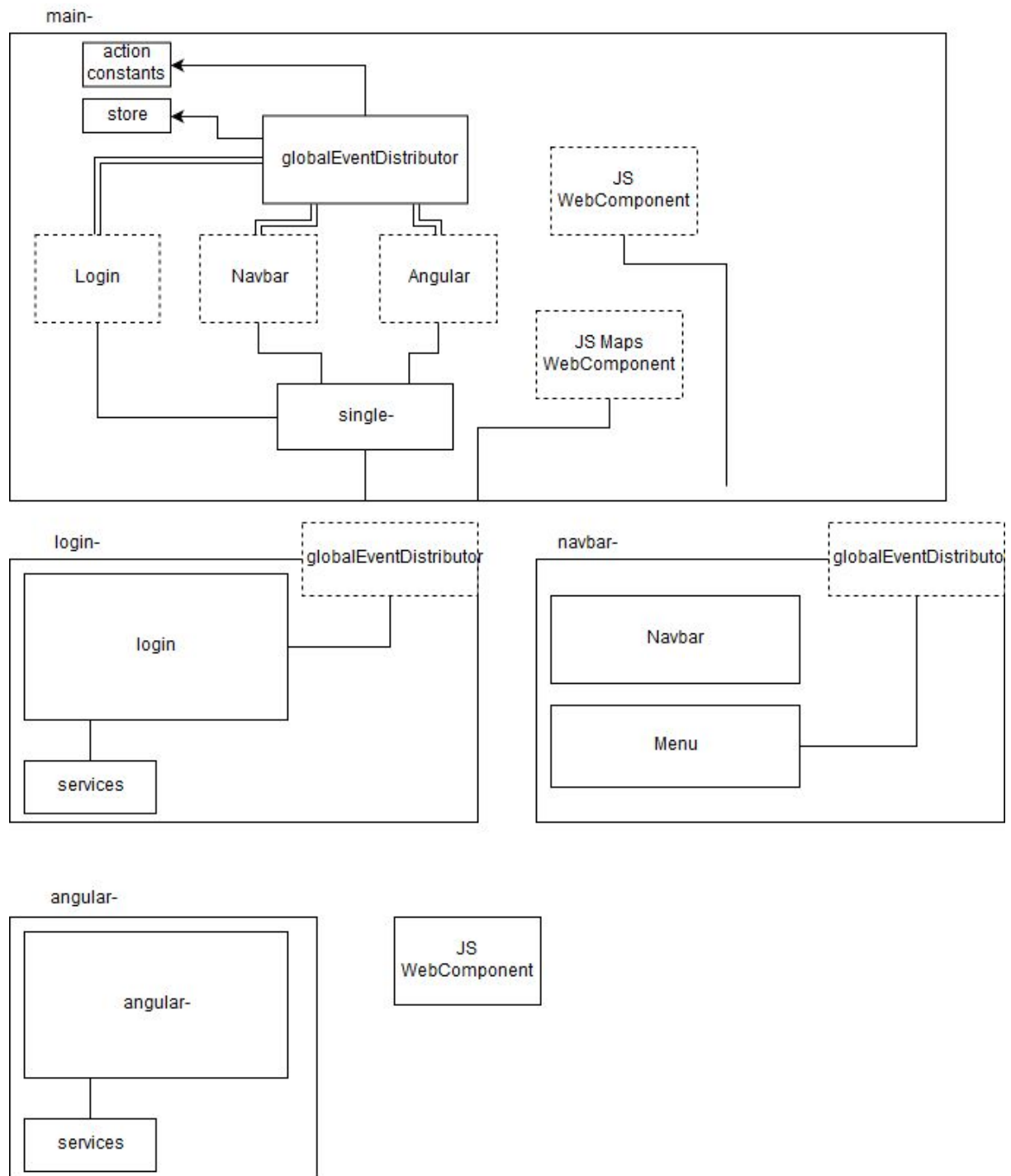
Repositorio

Dejamos un repositorio público en GitHub para que se pueda descargar la aplicación y seguir los pasos desde el readme para levantarlo.

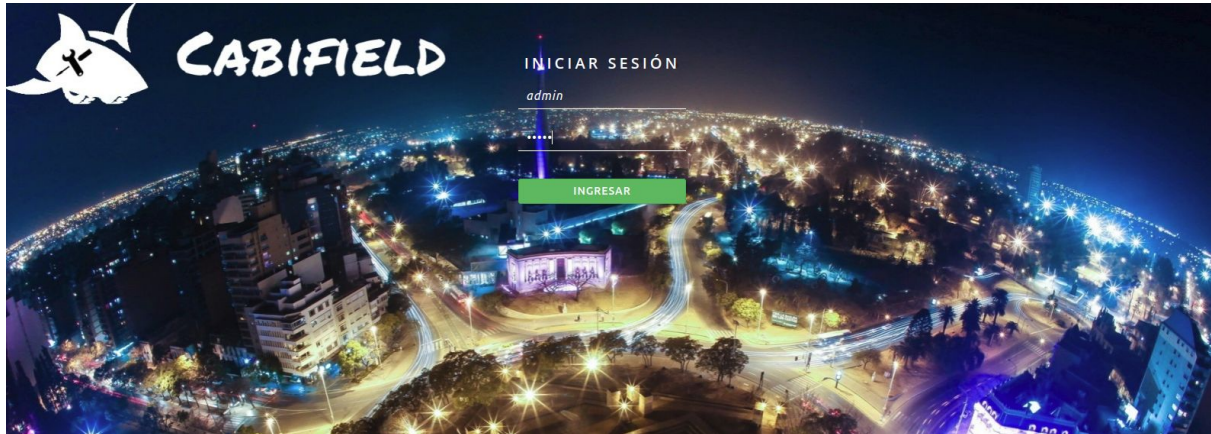
<https://github.com/emiliobucha/microfrontend-dummies>

En el Readme.md del mismo se encuentran la serie de comandos a seguir para levantar el proyecto

Arquitectura a implementar



Capturas finales.







 CabiField

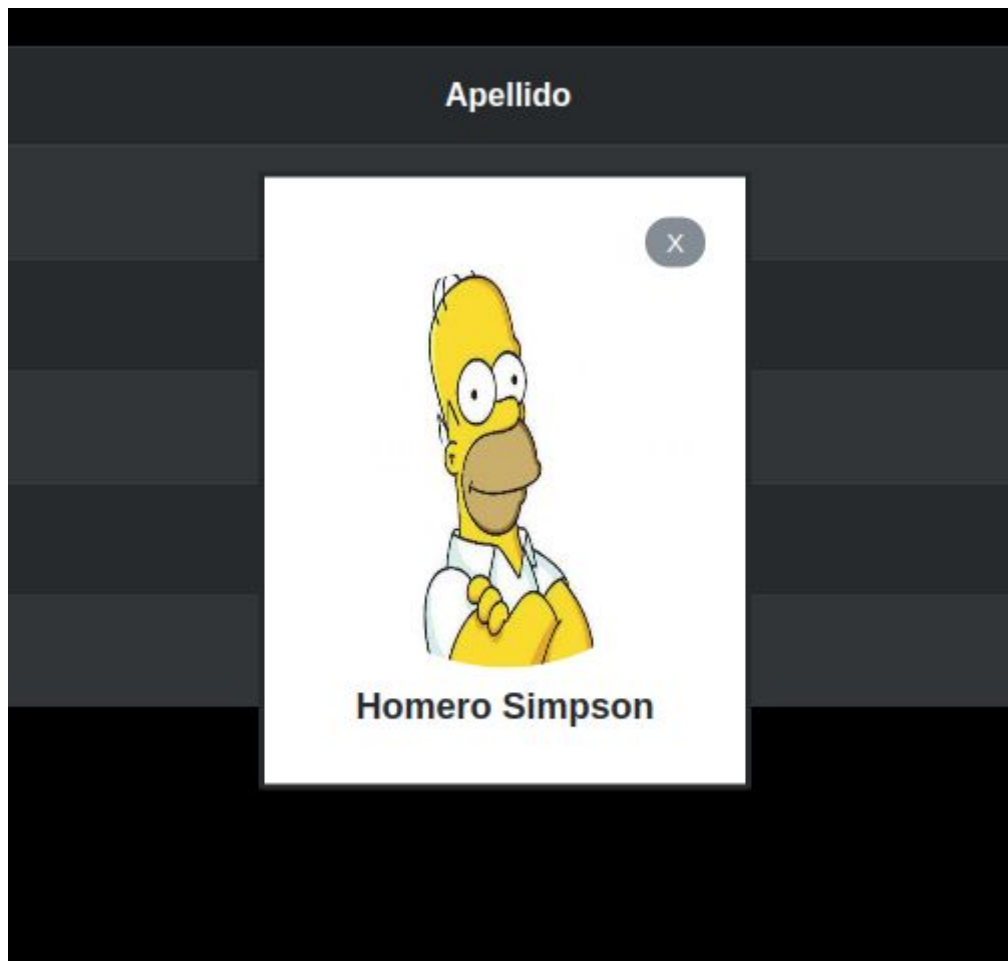
VUE

Hola, admin ▾

REACT

#	Nombre	Apellido	
0	Homero	Simpson	
1	Marjorie	Bouvier	
2	Lisa Marie	Simpson	
3	Maggie	Simpson	
4	Bart	Simpson	

Angular



Aplicación principal

Crear main-app

```
npx create-react-app main-app  
cd main-app
```

Instalar single-spa

```
npm i --save single-spa
```

App.js

En la aplicación principal vamos a ver un archivo que se generó, llamado App.js. Al cual le vamos a dar un formato de .jsx pero sin serlo, y declaramos un div con el id="react-app".

Es en este archivo en el que se va a renderizar cada una de las subaplicaciones que se crearán, mediante el id con el que se registran las subaplicaciones. Podemos definir a la main-app como el contenedor principal en el que se montaran las demás aplicaciones.

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {

  constructor() {
    super();
  }

  render() {
    return (
      <div>
        <div>
          <div>
            <div id="react-app"/>
          </div>
        </div>
      </div>
    );
  }
}

export default App;
```

Carpeta apps

En /src creamos la carpeta /apps

Creamos un archivo register-react-app.js con el siguiente contenido

```
import * as singleSpa from "single-spa";
import {matchingPathname, runScript} from "./utils";

const loadReactApp = async () => {
  await runScript('http://localhost:3002/static/js/main.js');
  return window.reactApp;
};
```

```
export const registerReactApp = () => {
  singleSpa.registerApplication('react-app', loadReactApp,
    matchingPathname(['/react', '/']));
};
```

Tener en cuenta que el main.js que se debe llamar debe estar dado por una subaplicación con un webpack customizado que se explicará más adelante, para que sea un solo llamado.

Utils

Donde tenemos en el método runScript debemos llamar la url donde se encuentra hosteado de manera abierta el main.js de la aplicación

Las utils se encuentran en el repositorio, runScript se encarga de cargar en un script del html principal la importación de la subaplicación

```
export const runScript = async (url) => {
  return new Promise((resolve, reject) => {
    const script = document.createElement('script');
    script.src = url;
    script.onload = resolve;
    script.onerror = reject;

    const firstScript = document.getElementsByTagName('script')[0];
    firstScript.parentNode.insertBefore(script, firstScript);
  });
};
```

Mientras que matchingPathname se encarga de verificar si la ruta actual permite renderizar o no la aplicación

```
export const matchingPathname = (pathnames) =>
  (location) =>
    pathnames.some(pathname => location.pathname === pathname);
```

Index.js

En index.js de main-app se importa la librería single-spa

```
import * as singleSpa from 'single-spa';
```

Llamamos a nuestro método de registerReactApp() y singleSpa.start(), quedándonos de la siguiente manera

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import * as singleSpa from 'single-spa';
import { registerReactApp } from './apps/register-react-app';

ReactDOM.render(<App />, document.getElementById('root'));

registerReactApp();
singleSpa.start();

serviceWorker.unregister();
```

Creación subaplicación Login

La subaplicación Login la implementaremos con tecnología react en la última versión la 16.8.6 al igual que la main-app.

Crear la aplicación

```
npx create-react-app login-app-react
cd login-app-react
```

Instalar single-spa para react

Esta librería auxiliar nos ayudara a implementar las funciones del ciclo de vida de las aplicaciones registradas en single-spa y poder usarla en react.

```
npm i --save single-spa-react
```

Ciclo de vida SingleSpaReact

Una vez instalada la librería debemos modificar el archivo index.js que se crea por defecto en /login-app-react/src/

Este archivo index.js permitirá montar, desmontar y cargar el bootstrap de la aplicación login-app-react en la aplicación principal(main-app) ya que se descargara como un módulo en el navegador.

Archivo index.js con su modificación:

```
import React from 'react';
import ReactDOM from 'react-dom';
import singleSpaReact from 'single-spa-react';
import rootComponent from './App';

const reactLifecycles = singleSpaReact({
  React,
  ReactDOM,
  rootComponent,
  domElementGetter: () => document.getElementById('login-app-react')
});

export function bootstrap(props) {
  return reactLifecycles.bootstrap(props);
}

export function mount(props) {
  return reactLifecycles.mount(props);
}

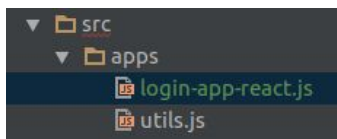
export function unmount(props) {
  return reactLifecycles.unmount(props);
}
```

La función singleSpaReact(opciones) recibe parametros , algunas opciones son requeridas y otras opciones:

- React y ReactDOM son requeridas: Son los objetos principales de React.
- RootComponent: Es el componente padre de la aplicación que se renderiza.
- domElementGetter: Esta función es opcional, en nuestro ejemplo la utilizamos para definir el id con el que se identificara la aplicación login-app-react para poder montarse y desmontarse en la aplicación principal.

Registrar subaplicación login

Luego de crear el ciclo de vida de la aplicación login con single-spa, debemos registrar esta aplicación(subaplicacion) en la aplicación principal para que pueda ser montada. Para esto debemos ir a la aplicación principal y en la carpeta src/apps/ crear un nuevo archivo para declarar la función de registro de la app, en nuestro caso le llamamos login-app-react.js.



En este archivo se debe llamar a la función `registerApplication()` de la librería `single-spa`. Esta función es la más importante de la librería, ya que aquí se registra cualquier aplicación que creamos.

```
import * as singleSpa from "single-spa";
import {matchingPathname, runScript} from "./utils";
import {userConstants} from '../store/constants';

const loadReactApp = async () => {

  await runScript('http://localhost:3002/static/js/main.js');
  return window.loginApp;
};

export const registerLoginAppReact = (globalEventDistributor) => {
  const customProps = {
    globalEventDistributor: globalEventDistributor,
    constants: userConstants
  };
  singleSpa.registerApplication('login-app-react', loadReactApp,
    matchingPathname(['/login', '/']), customProps);
};
```

Los argumentos que recibe esta función son:

Nombre de la aplicación: nombre con la que Single-spa registrará y hará referencia a esta aplicación, y se utilizará en el index principal.

Función de carga: aquí se le pasa la ruta del servidor en el que está corriendo la aplicación seguido de la carpeta en la que se generó el build de la aplicación `login-app-react`.

Función que compara ruta: Debe ser una función pura. La función se llama con `window.location` como primer argumento y debe devolver un valor verdadero siempre que la aplicación esté activa.



Si la url es la misma que se pasa por parámetro, la función devuelve `true` y renderiza la subaplicación en la aplicación principal mediante el `id="login-app-react"`.

CustomProps: Son propiedades personalizadas que se pasan a la aplicación durante cada método del ciclo de vida.

Agregar tags de subaplicación en aplicación principal

En la aplicación principal dentro src/ en el archivo App.js se debe agregar la etiqueta de la aplicación que se registró para que pueda mostrarse en la página principal.

```
import React, { Component } from 'react';
import './App.css';

class App extends Component {

  constructor() {
    super();
  }

  render() {
    return (
      <div>
        <div>
          <div>
            <div id="react-app"/>
            <div id="login-app-react"/>
          </div>
        </div>
      </div>
    );
  }
}

export default App;
```

Llamar al registerLoginAppReact e iniciar single-spa

Dentro de /src/index.js se debe llamar a la función que registra la subaplicación login e inicializar single-spa mediante la función start().

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
```

```

import * as serviceWorker from './serviceWorker';
import * as singleSpa from 'single-spa';
import { registerLoginAppReact } from './apps/login-app-react';
import { storeInstance } from './store/reducers/store';
import { GlobalEventDistributor } from './store/globalEventDistributor';
import { Provider, connect } from 'react-redux';
import { userConstants } from './store/constants';

ReactDOM.render(
  <Provider store={storeInstance}>
    <App />
  </Provider>,
  document.getElementById('root')
);

const globalEventDistributor = new GlobalEventDistributor();

globalEventDistributor.registerStore(storeInstance, userConstants.STORE_AUTH);

registerLoginAppReact(globalEventDistributor);
singleSpa.start();

serviceWorker.unregister();

```

Utilización de webpack en login

La librería single-spa no provee un webpack por defecto para React, por lo que deberemos generar un archivo webpack personalizado para nuestra aplicación.

Lo más importante en la configuración del webpack son las opciones: entry, output y las rules dentro de module.

Entry:

```
entry: [require.resolve('./polyfills'), paths.appIndexJs],
```

En la opción entry se le debe pasar el archivo en el que se crea el ciclo de vida de la subaplicación single-spa de react

a través del método singleSpaReact(), en nuestro caso es el archivo index.js que se encuentra en /src/index.js.

Output

```
output: {
  // The build folder.
  path: paths.appBuild,
  // Generated JS file names (with nested folders).
  // There will be one main bundle, and one file per asynchronous
  chunk.
  // We don't currently advertise code splitting but Webpack supports
  it.
  filename: 'static/js/[name].js',
  chunkFilename: 'static/js/[name].chunk.js',
  // We inferred the "public path" (such as / or /my-project) from
  homepage.
  publicPath: publicPath,
  // Point sourcemap entries to original disk location (format as URL
  on Windows)
  devtoolModuleFilenameTemplate: info =>
    path
      .relative(paths.appSrc, info.absoluteResourcePath)
      .replace(/\\/g, '/'),
  library: "loginApp",
  libraryTarget: "window"
},
```

Está es una de las opciones más importante en el archivo webpack.

Path: Se debe ingresar un nombre con el que se creara la carpeta contenedora de los archivos cuando se realice el build.

Filename: Nombre de los archivos que se van a generar incluyendo la carpeta contenedora

PublicPath: Agregar la ruta pública desde la página de inicio.

Library: está opción es muy importante en el uso de nuestra aplicación ya que con el nombre que se ingresa, es con la que se utiliza en la aplicación principal para cargar el archivo que se cargue en filename, en este caso el main.js.

LibraryTarget: Aquí definimos el tipo de la library que vamos a utilizar para luego llamarla desde la aplicación principal, en nuestra caso usamos el window.

Creación subaplicación navbar

Para completar con la utilización de los framework de frontends más utilizados en la actualidad, el componente navbar lo implementamos con VueJs en la última versión 2.6.10.

Crear aplicación

```
vue create navbar-app-vue
cd navbar-app-vue
```

Instalar single-spa-vue

Al igual que en las anteriores aplicaciones se debe instalar la librería de single-spa que de soporte al uso del framework VueJs.

Para esto existen dos formas de instalar esta librería.

- Manualmente: utilizando el npm con el siguiente comando.

```
npm install --save single-spa-vue
```

- Vue CLI: está es la alternativa que utilizamos nosotros ya que estamos usando el CLI para crear,deployar el proyecto.

```
vue add single-spa
```

El comando add single-spa a través del CLI lo que hace es:

- Modifica la configuración de webpack para que funcione la aplicación con single-spa
- Modifica el archivo main.js para que funcione con single-spa
- Instala single-spa-vue

Ciclo de vida SingleSpaVue

Cómo instalamos la librería con el CLI de vue, por defecto nos modifica el archivos main.js y nos crea los métodos y la construcción del ciclo de vida de single-spa-vue

```
import Vue from 'vue'
```

```

import App from './App.vue'
import 'bootstrap/dist/css/bootstrap.css'
import 'bootstrap-vue/dist/bootstrap-vue.css'

import singleSpaVue from 'single-spa-vue';

import { NavbarPlugin } from 'bootstrap-vue'
Vue.use(NavbarPlugin)

Vue.config.productionTip = false

const vueLifecycles = singleSpaVue({
  Vue,
  appOptions: {
    el: '#navbar-app-vue',
    render: h => h(App)
  }
});

export function bootstrap(props) {
  return vueLifecycles.bootstrap(props);
}

export function mount(props) {
  createDomElement();
  return vueLifecycles.mount(props);
}

export function unmount(props) {
  return vueLifecycles.unmount(props);
}

function createDomElement() {
  // Make sure there is a div for us to render into
  let el = document.getElementById('navbar-app-vue');

  if (!el) {
    el = document.createElement('div');
    el.id = 'navbar-app-vue';
    document.body.appendChild(el);
  }
  return el;
}

```

Los métodos que se crean por defecto son: mount(), unmount() y bootstrap().

En esta prueba de concepto se observa que estos métodos reciben como parámetro props, éstas son propiedades que se envían desde la aplicación principal y las recibe la subaplicación navbar para ser utilizadas. En nuestro caso usamos los datos del usuario que se envían desde la subaplicación Login mediante el globalEventDistributor ([Comunicación y GlobalEventDistributor](#))

Una función que agregamos es createDomElement() en el que indicamos el id con el que se montara en la aplicación principal.

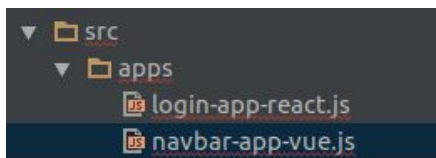
Cuando se crea el ciclo de vida de single-spa-vue todas las opciones se pasan a través del del parámetro opts cuando se llama a singleSpaVue(opts).

Las siguientes opciones:

- Vue (obligatorio): Es el objeto principal Vue, que está disponible a través del import "vue"
- appOptions: Este objeto se utiliza para crear una instancia de la aplicación Vue. Hay que tener en cuenta que si no se crea un elemento "el" en appOptions, se creará un div y se agregara al DOM como contenedor predeterminado para la aplicación Vue.

Registrar subaplicación navbar

Debemos registrar esta aplicación(subaplicacion) en la aplicación principal para que pueda ser montada. Para esto debemos ir a la aplicación principal y en la carpeta src/apps/ crear un nuevo archivo(navbar-app-vue.js) en el que se crea una función para registrar la subaplicación.



En el archivo navbar-app-vue.js se debe llamar a la función registerApplication() de la librería single-spa.

```
import * as singleSpa from "single-spa";
import {matchingPathname, runScript} from "../utils";
import {userConstants} from '../store/constants';

const loadVueApp = async () => {
  await runScript('http://localhost:3004/build.js');
  return window.navbarvueApp;
};
```

```
export const registerNavbarAppVue = (globalEventDistributor) => {
  const customProps = {
    globalEventDistributor: globalEventDistributor,
    constants: userConstants
  };
  singleSpa.registerApplication('navbar-app-vue', loadVueApp,
    matchingPathname(['/navbar', '/home']), customProps);
};
```

La implementación de la función registerApplication implementa los mismos conceptos que se definieron en [Registrar subaplicación login](#) lo que se debe modificar es el puerto que se definió para la subaplicación navbar, la url que se cargará en la aplicación principal es <http://localhost:3004/build.js>.

Agregar tags subaplicación en aplicación principal

En la aplicación principal dentro src/ en el archivo App.js se debe agregar la etiqueta de la aplicación que se registró para que pueda mostrarse en la página principal.

```
render() {
  return (
    <div>
      <div>
        <div>
          <div id="react-app"/>
          <div id="navbar-app-vue"/>
          <div id="angularapp"/>
          <div id="login-app-react"/>
        </div>
      </div>
    </div>
  );
}
```

Llamar al registerNavbarAppVue

Al igual que la subaplicación login luego de crear el método para cargar archivo que hemos buildeado dentro de la aplicación principal hay llamar al método dentro de /src/index.js. Para esto llamamos a la función registerNavbarAppVue(customProps), con las propiedades personalizadas.

```

import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import * as singleSpa from 'single-spa';
import { registerLoginAppReact } from './apps/login-app-react';
import { storeInstance } from './store/reducers/store';
import { GlobalEventDistributor } from './store/globalEventDistributor';
import { Provider, connect } from 'react-redux';
import { userConstants } from './store/constants';
import { registerAngularApp } from './apps/register-angular-app';
import { registerNavbarAppVue } from './apps/navbar-app-vue';

ReactDOM.render(
  <Provider store={storeInstance}>
    <App />
  </Provider>,
  document.getElementById('root')
);

const globalEventDistributor = new GlobalEventDistributor();

globalEventDistributor.registerStore(storeInstance, userConstants.STORE_AUTH);

registerAngularApp();
registerLoginAppReact(globalEventDistributor);
registerNavbarAppVue(globalEventDistributor);
singleSpa.start();

serviceWorker.unregister();

```

Webpack navbar

Al igual que en login, utilizamos un custom webpack, teniendo en cuenta las mismas características principales que planteamos en la app de Login, solo que cambiamos algunas “rules” para que funcionen los archivos .vue y el bootstrap vue y en “output” definimos el nombre de library : ‘navbarvueApp’.

Mejoras pendientes

En futuras actualizaciones del repositorio

<https://github.com/emiliobucha/microfrontend-dummies> nos queda mejorar el buildeo e inicialización de las aplicaciones react y vue, implementandolo mediante los CLI de las respectivas tecnologías. De esta forma no se tendrá que invertir tiempo en configurar el webpack para cada framework y nos podremos enfocar en la funcionalidad de cada aplicación que desarrollemos.

Creación subaplicación Grilla

La subaplicación grilla la implementaremos con tecnología Angular en la última versión la 8.0.3

Crear la aplicación

```
ng new angularapp --routing --prefix angularapp
cd angularapp
```

Instalar single-spa para angular

```
ng add single-spa-angular@beta
```

Este último comando realiza las siguientes acciones:

- instalar single-spa-angular
- Genera un archivo main.single-spa.ts en la ruta ./src.
- Genera un archivo single-spa-props.ts en ./src/single-spa.
- Genera archivo asset-url.ts en ./src/single-spa
- Agregue un script npm run build: single-spa en package.json.
- Agregue un script npm run serve: single-spa package.json.

Lo siguiente a hacer es configurar las rutas para que funcione single-spa esto debe hacerse manualmente agregando al app-routing.module.ts el providers

```
providers: [  
  { provide: APP_BASE_HREF, useValue: '/' }  
]
```

Componente Grilla

Creamos el componente grilla

```
ng g c grilla
```

El siguiente comando crea los archivos necesarios para el funcionamiento del componente



Debemos añadir el path en el array de Routes

```
{path: '**', component: GrillaComponent }
```

de esta forma se garantiza que cuando un single-spa realiza transacciones entre rutas, la aplicación angular no muestre un 404, o genere un error.

creamos un servicio genérico de manera simple que provee de información para poder cargar datos en la grilla y poder utilizar el WebComponent JS para mostrar una tarjeta de personaje



Registro en Aplicación Principal

Debemos registrar el proyecto de angular en la aplicación principal para eso vamos hasta la carpeta que contiene la aplicación principal y dentro de src/apps creamos un archivo js con la siguiente información

```
import * as singleSpa from "single-spa";
import {matchingPathname, runScript} from "./utils";

const loadAngularApp = async () => {
  await runScript('http://localhost:4200/main.js');
  return window.app3;
};

export const registerAngularApp = () => {
  singleSpa.registerApplication('angularapp', loadAngularApp,
    matchingPathname(['/angularapp', '/']));
};
```

nos dirigimos al index.js de la aplicación principal e importamos el metodo

registerAngularApp

quedando de la siguiente forma

```
import React from 'react';
```



```
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import * as singleSpa from 'single-spa';
import { registerAngularApp } from './apps/register-angular-app';

ReactDOM.render(<App />, document.getElementById('root'));

registerAngularApp();
singleSpa.start();

serviceWorker.unregister();
```

Debemos agregar en el main.single-spa.ts en la configuración de singleSpaAngular la siguiente línea

```
domElementGetter: () => document.getElementById('angularapp'),
```

quedando de esta forma

```
const lifecycles = singleSpaAngular({
  domElementGetter: () => document.getElementById('angularapp'),
  bootstrapFunction: singleSpaProps => {
    singleSpaPropsSubject.next(singleSpaProps);
    return platformBrowserDynamic().bootstrapModule(AppModule);
  },
  template: '<angularapp-root />',
  Router,
  NgZone: NgZone,
  AnimationEngine: AnimationEngine,
});
```

Y dentro del render() de app.js insertamos el div con el id del nombre de la aplicación definida en el main.single-spa.ts de angular

```
render() {
  return (
    <div>
      <div>
        <div>
          <div id="react-app"/>
          <div id="angularapp"/>
        </div>
      </div>
    </div>
  );
}
```

```

        </div>
    </div>
  );
}

```

Principales inconvenientes single-spa-angular

```

✖ ▶ Uncaught angularapp: Application 'angularapp' died in      single-spa.min.js:189
  status UNMOUNTING: Cannot read property 'injector' of undefined
  at http://localhost:4200/main.js:105293:53

✖ ▶ Uncaught angularapp: Application 'angularapp' died in      single-spa.min.js:189
  status SKIP_BECAUSE_BROKEN: In this configuration Angular requires Zone.js
  at new NgZone (http://localhost:4200/main.js:75161:19)
  at getNgZone (http://localhost:4200/main.js:76190:13)
  at PlatformRef.bootstrapModuleFactory
  (http://localhost:4200/main.js:76033:24)
  at http://localhost:4200/main.js:76108:31

```

Para solucionar el siguiente error primero debemos importar NgZone y core-js al main.single-spa.ts por alguna razón esto no lo realiza de manera automática

```

import 'core-js/proposals/reflect-metadata';
import 'zone.js/dist/zone';

```

quedando finalmente las importaciones de este archivo así

```

import 'core-js/proposals/reflect-metadata';
import 'zone.js/dist/zone';
import { enableProdMode, NgZone } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { Router } from '@angular/router';
import { ɵAnimationEngine as AnimationEngine } from
'@angular/animations/browser';
import { AppModule } from './app/app.module';
import { environment } from './environments/environment';
import singleSpaAngular from 'single-spa-angular';
import { singleSpaPropsSubject } from './single-spa/single-spa-props';

```

Otro de los inconveniente que tuvimos fue el siguiente

```
✖ ▶ The loading function for single-spa application 'angularapp' single-spa.min.js:601
resolved with the following, which does not have bootstrap, mount, and unmount
functions <div id="angularapp"></div>

⚠ ▶ While LOADING_SOURCE_CODE, 'angularapp' rejected its lifecycle single-spa.min.js:219
function promise with a non-Error. This will cause stack traces to not be accurate.

✖ ▶ Uncaught angularapp: Application 'angularapp' died in status zone-evergreen.js:172
LOADING_SOURCE_CODE: "does not export an unmount function or array of functions"
at m (http://localhost:5000/static/js/2.77686a9a.chunk.js:1:3588)
at f (http://localhost:5000/static/js/2.77686a9a.chunk.js:1:2834)
at http://localhost:5000/static/js/2.77686a9a.chunk.js:1:12005
```

esto se debe a el nombre de la aplicación que nosotros le damos al crearla utilizando --prefix que sirve para poder identificar las diferentes aplicaciones que pueden existir en la implementación de MicroFrontend, El webpack de single-spa-angular al general el build lo configura por defecto con el nombre de App3

```
output:
  { futureEmitAssets: true,
    path:
      '/home/cids/microfrontend-dummies/apps/angularapp/dist/angularapp',
    publicPath: 'http://localhost:4200/',
    filename: '[name]-es2015.[chunkhash:20].js',
    crossOriginLoading: false,
    library: 'app3',
    libraryTarget: 'umd' },
  watch: false,
  watchOptions: { poll: undefined },
  performance: { hints: false },
  module:
    { strictExportPresence: true,
      rules:
```

aca podemos ver un ejemplo cuando genera el build asigna como nombre (library:'app3') por defecto, lo cual tenemos dos alternativas o cambiarle el nombre a la aplicación cuando la registramos por app3 esto significa un inconveniente al tener más de una aplicación en angular perdiendo así la referencia a cual se está llamando.

```
const loadAngularApp = async () => {
  await runScript('http://localhost:4200/main.js');
  return window.app3;
};
```

La otra alternativa es entrar a node_module/single-spa-angular/lib/webpack/index.js

```
const singleSpaConfig = {
  output: {
    library: 'app3',
    libraryTarget: 'umd',
  },
  externals: {
    'zone.js': 'Zone',
```

y cambiarle el nombre por uno diferente por ejemplo

```
const singleSpaConfig = {
  output: {
    library: 'angularapp',
    libraryTarget: 'umd',
  },
  externals: {
    'zone.js': 'Zone',
  },
};
```

y dentro de la aplicación principal register-angular-app.js cambiarle el window.app3 por window.angularapp o la aplicación que queramos registrar.

```
const loadAngularApp = async () => {
  await runScript('http://localhost:4200/main.js');
  return window.angularapp;
};
```

pero esto conlleva una complicación a la hora de eliminar el node_module o al descargarlo de un repositorio y realizar un npm install el valor establecido en node_module/single-spa-angular/lib/webpack/index.js se restablece con app3 tirando de nuevo el error ya que al registrar la aplicación queda con window.angularapp

WebComponent JS

El Web Component realizado en JS se implementa de forma sencilla en un solo archivo .js servido en un servidor node express.

La implementación del mismo se basa en la declaración de una nueva class heredada de HTMLElement

```
class Modal extends HTMLElement
```

Getters - Setters

Luego dentro de la misma la declaración de getters y setters de los atributos de la misma

```
get visible() {
  return this.hasAttribute("visible");
}
set visible(value) {
  if (value) {
    this.setAttribute("visible", "");
  } else {
    this.removeAttribute("visible");
  }
}
```

```

        this.removeAttribute("visible");
    }
}
get foto() {
    return this.getAttribute('foto');
}
set foto(value) {
    this.setAttribute('foto', value);
}
get nombre() {
    return this.getAttribute('nombre');
}
set nombre(value) {
    this.setAttribute('nombre', value);
}
}

```

Ciclo de Vida

Luego declaramos los métodos del ciclo de vida del Custom Element que necesitamos

```

constructor() {
    super();
}

connectedCallback() {
    this._render();
    this._attachEventHandlers();
}

static get observedAttributes() {
    return ["visible", "foto", "nombre"];
}

attributeChangedCallback(name, oldValue, newValue) {
    console.log(newValue);
    if (name === "nombre" && this.shadowRoot) {
        this.shadowRoot.getElementById("nombre").textContent =
newValue;
    }
    if (name === "foto" && this.shadowRoot) {
        this.shadowRoot.getElementById("foto").src = newValue;
    }
    if (name === "visible" && this.shadowRoot) {
        if (newValue === null) {

```

```

this.shadowRoot.querySelector(".wrapper").classList.remove("visible");
    this.dispatchEvent(new CustomEvent("close"));
  } else {

this.shadowRoot.querySelector(".wrapper").classList.add("visible");
    this.dispatchEvent(new CustomEvent("open"))
  }
}
}

```

Render

Y por último la declaración del método `this._render()`, nos determina la creación de un `<div>` donde dentro se le inyecta y declara el HTML y los estilos del componente. (Se resumen los estilos para no alargar la explicación)

```

_render() {
  const wrapperClass = this.visible ? "wrapper visible" :
  "wrapper";
  const container = document.createElement("div");
  container.innerHTML = `
    <style>
      .wrapper {
        position: fixed;
      }

      .visible {
        opacity: 1;
      }
    <div class='${wrapperClass}'>
      <div class='modal'>
        <div style='text-align: right'>
          <button class='close'>X</button>
        </div>
        <div>
          <div style='text-align: center'>
            <img id='foto' src='${this.foto}' class='imgRedonda' height='200'>
          </div>
        </div>
        <div class='tarjeta-content'>
          <div>
            <span id='nombre'>${this.nombre? this.nombre: '-'}</span>
          </div>
        </div>
      </div>
    </div>
  `
}

```

```

    </div>
  `;
  const shadowRoot = this.attachShadow({ mode: 'open' });
  shadowRoot.appendChild(container);
}

```

Declaración de custom element

y finalmente se declara el custom element dentro de window

```

window.customElements.define('tarjeta-foto', Modal);

```

Utilización

Luego para poder ser utilizado este Web Component dentro de la arquitectura de microfrontend planteada, se declaró la llamada a este javascript dentro del index.html principal de la aplicación principal.

```

<!--WebComponents-->
<script src="http://localhost:5000/tarjeta-foto.js"></script>

```

Para luego ser utilizado dentro de Angular y poder utilizar el tag del Custom Element, se declara en el app.module.ts principal

```

import { NgModule, CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';

```

y se agregan al @NgModule

```

schemas: [ CUSTOM_ELEMENTS_SCHEMA ]

```

y se utiliza el tag dentro del html de la forma

```

<tarjeta-foto></tarjeta-foto>

```

seteandole los atributos a través del código

```

abrirTarjeta(personaje) {
  if (!personaje) {
    personaje = {nombre: 'Homer', apellido: 'Simpson', foto:
'https://media.canalnet.tv/2018/08/Homer-Simpson.jpeg'};
  }

  const modal = document.querySelector('tarjeta-foto');

```

```
const nombreCompleto = personaje.nombre + ' ' + personaje.apellido;
modal.setAttribute('nombre', nombreCompleto);
modal.setAttribute('foto', personaje.foto);
modal.setAttribute('visible', 'true');

}
```

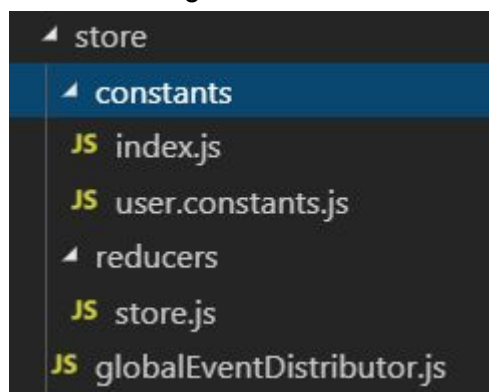

Comunicación y GlobalEventDistributor

Para realizar la comunicación y almacenamiento de información entre los distintos componentes, se puede implementar un “bus” de eventos al cual está suscrito un array de stores de Redux.

Hay muchos tutoriales de Redux en la web y no vamos a ahondar en esta herramienta, sino que se verá la configuración y comunicación de los distintos componentes.

En el proyecto main-app se declaró un store principal para poder almacenar datos de sesión del usuario, los cuales muy usualmente se utilizan desde varios componentes.

Creamos la siguiente estructura de carpetas



Donde nos encontramos con:

1. Constants: las cuales son constantes de las acciones y stores de Redux.
2. Reducers: se encuentra el único store actualmente declarado y utilizado, en este caso para guardar los datos del login.
3. GlobalEventDistributor: es el objeto principal donde se almacenan todos los stores en un array y realiza el dispatch de la acción a todos los stores quienes serán los encargados, como si fuera un evento, de ejecutar lo que corresponda si está declarado dentro de sí.

Redux Main-App

Para comenzar la posibilidad de utilizar este Store de la aplicación principal haremos una implementación sencilla de Redux.

Primero debemos instalar redux y react-redux, los mismos nos van a permitir crear el reducer. Ambos en la carpeta de la aplicación principal

```
npm i --save redux
npm i --save react-redux
```

Quedándonos las dependencias finales de nuestro package.json así

```
"dependencies": {
  "react": "^16.8.6",
  "react-dom": "^16.8.6",
  "react-redux": "^7.1.0",
  "react-scripts": "3.0.1",
  "redux": "^4.0.4",
  "single-spa": "^4.3.7"
},
```

En el index.js de la aplicación principal vamos a realizar la lógica de inicialización de este GlobalEventDistributor y declarar al store como provider.

El index.js final de la main-app quedaría así incluidos los registros de las apps.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import * as singleSpa from 'single-spa';
import { registerLoginAppReact } from './apps/login-app-react';
import { storeInstance } from './store/reducers/store';
import { GlobalEventDistributor } from './store/globalEventDistributor';
import { Provider } from 'react-redux';
import { userConstants } from './store/constants';
import { registerAngularApp } from './apps/register-angular-app';
import { registerNavbarAppVue } from './apps/navbar-app-vue';

ReactDOM.render(
  <Provider store={storeInstance}>
    <App />
  </Provider>,
  document.getElementById('root')
);

const globalEventDistributor = new GlobalEventDistributor();
globalEventDistributor.registerStore(storeInstance,
userConstants.STORE_AUTH);

registerAngularApp();
registerLoginAppReact(globalEventDistributor);
registerNavbarAppVue(globalEventDistributor);
singleSpa.start();
serviceWorker.unregister();
```

Store y Constants

El siguiente Store está pensado para guardar los datos del usuario una vez logueado.

```
import { userConstants } from '../constants';
import { createStore } from 'redux';

// User obtenido del localStorage para ver que esté logeado
let user = JSON.parse(localStorage.getItem('user'));

const initialState = user ? { loggedIn: true, user } : {};

// Reducer de Authentication
export function authentication(state = initialState, action) {
  switch (action.type) {
    case userConstants.LOGIN:
      return {
        loggedIn: true,
        user: action.user
      };
    case userConstants.LOGOUT:
      return { loggedIn: false, user: null };
    default:
      return state
  }
}

export const storeInstance = createStore(authentication);
```

Las constantes es tan solo un objeto de string paramétricos para las acciones de los stores.

```
// Constantes de acciones de login
export const userConstants = {
  LOGIN: 'LOGIN',
  LOGOUT: 'LOGOUT',
  STORE_AUTH: 'STORE_AUTH'
};
```

GlobalEventDistributor

Como fue nombrado anteriormente, este es el encargado de mantener los stores de las distintas apps y hacer de una especie de eventbus.

```
export class GlobalEventDistributor {

  constructor() {
    this.stores = [];
  }

  registerStore(store, name) {
    this.stores.push({...store, name});
    //console.log(this.stores);
  }

  dispatch(event) {
    this.stores.forEach((s) => s.dispatch(event));
    //this.stores.forEach(x=>console.log(x.getState()));
  }
}
```

Inyección de GlobalEventDistributor en aplicación de Login

```
import * as singleSpa from "single-spa";
import {matchingPathname, runScript} from "./utils";
import {userConstants} from '../store/constants';

const loadReactApp = async () => {

  await runScript('http://localhost:3002/static/js/main.js');
  return window.loginApp;
};

export const registerLoginAppReact = (globalEventDistributor) => {
  const customProps = {
    globalEventDistributor: globalEventDistributor,
    constants: userConstants
  };
  singleSpa.registerApplication('login-app-react', loadReactApp,
    matchingPathname(['/login', '/']), customProps);
}
```

```
};
```

Como ya se ha mostrado cuando se creó la aplicación de login, se inyectó de la forma anteriormente nombrada, esto permite mediante los customProps que declara Single-SPA que se le pasen atributos o instancias de clases como se desee y estas cuando se inicializan las aplicación son inyectadas en tiempo de ejecución.

Para el caso del Login en React (es de igual forma para cualquier aplicación de React) Como vimos anteriormente, el index.js de esta aplicación de React se declaran los método que Single-SPA necesita para instanciar una nueva aplicación.

```
export function bootstrap(props) {  
  return reactLifecycles.bootstrap(props);  
}  
  
export function mount(props) {  
  return reactLifecycles.mount(props);  
}  
  
export function unmount(props) {  
  return reactLifecycles.unmount(props);  
}
```

A través de los parámetros props ingresan los customProps declarados en la aplicación principal para instanciar esta subaplicación de React.

Por lo tanto en el rootComponent de la aplicación, en este caso es App.js vemos que ingresan las props como parámetro y son enviadas al componente Login.

```
import React from 'react';  
import logo from './logo.svg';  
import './App.css';  
import './styles/bootstrap/bootstrap.min.css';  
  
import { Login } from './Login'  
  
function App(props) {  
  return (  
    <Login {...props} />  
  )  
}  
  
export default App;
```

Donde en Login.js dentro de su constructor también recibe estas props y las podemos manipular.

```
constructor(props) {
```

```

    super(props);
    this.state = {
      username: '',
      password: '',
      submitted: false,
      globalEventDistributor: this.props.globalEventDistributor,
      constants: this.props.constants,
    };
    this.login = this.login.bind(this);
    this.handleChange = this.handleChange.bind(this);
  }

```

Por ejemplo en este login mockeado se puede utilizar la action declarada en el store

```

if (username === this.user.username && password === this.user.password) {
  ToastsStore.success("Bienvenido: " + this.user.username);
  globalEventDistributor.dispatch({type: constants.LOGIN, user:
this.user});
  localStorage.setItem('user', JSON.stringify(this.user));
} else {
  ToastsStore.error('Usuario y/o contraseña incorrecto');
}

```

Inyección de GlobalEventDistributor en aplicación de Navbar

De igual manera que en React se declara como customProps

```

export const registerNavbarAppVue = (globalEventDistributor) => {
  const customProps = {
    globalEventDistributor: globalEventDistributor,
    constants: userConstants
  };
  singleSpa.registerApplication('navbar-app-vue', loadVueApp,
matchingPathname(['/navbar', '/home']), customProps);
};

```

Dentro del main.js de la aplicación de Vue de Navbar vemos también que debemos tener qué en cuenta que se envían las props como parámetro

```

export function bootstrap(props) {
  return vuellifecycles.bootstrap(props);
}

export function mount(props) {
  createDomElement();
  return vuellifecycles.mount(props);
}

```

```
export function unmount(props) {
  return vueLifecycles.unmount(props);
}
```

Estas son inyectadas por Single-SPA en el atributo `this.$parent` de la aplicación de Vue, donde en `App.vue` lo implementamos de la siguiente forma enviando esa información al componente `Navbar.Vue`

```
<template>
  <div id="navbar-app-vue">
    <Navbar v-bind:customProps="this.customProps" />
  </div>
</template>

<script>
import Navbar from './components/Navbar.vue'

export default {
  name: 'navbar-app-vue',
  components: {
    Navbar
  },
  data(){
    return {
      customProps: this.$parent
    }
  }
}
</script>

<style>
#navbar-app-vue {}
</style>
```

En el script del componente `Navbar`, recibe como prop esta propiedad llamada `customProps` y puede ser utilizado de la siguiente manera.

```
<script>
export default {
  name: 'Navbar',
  data() {
```

```

        return {
            ged: this.customProps.globalEventDistributor,
            constants: this.customProps.constants,
            userInfo: null,
        }
    },
    props: ['customProps'],
    created() {
        const authStore = this.ged.stores.find(x => x.name ===
this.constants.STORE_AUTH);
        this.userInfo = authStore.getState().user;
    },
    methods: {
        logout() {
            this.ged.dispatch({
                type: this.constants.LOGOUT
            });
            delete localStorage['user'];
            this.navigateTo('/login');
        },
        navigateTo(url) {
            window.history.replaceState(null, null, url);
        }
    }
}
</script>

```

Tener en cuenta que se utiliza dentro del data() para que pueda ser renderizado por una interpolación en el html.

```

<div class="name">{{this.userInfo.nombre}}</div>
<div class="job">{{this.userInfo.mail}}</div>

```

En el método created() vemos que se llama a este globalEventDistributor para poder obtener un Store en particular, donde se lo llama por su atributo nombre declarado dentro de las constantes e inicializado en la aplicación principal y en base a este se obtienen los datos del usuario.