Inter-app communication

1. Communication between Container and Payment-list app

The mount function we have created in our micro apps can receive objects from the container if we provide a second property.

In our applications we are going to be using Axios, and so we might want to set a property in the container such as the remote server name, and pass this to the micro-apps that need it.

Let's implement this as our first example.

In payments-list,

- Convert the entry point Component (PaymentsList) to accept properties

```
const PaymentsList = (props: PaymentsListProps) : JSX.Element => {
    return <>
    <h2>This is the payments list</h2>
    <p>Data will be obtained from {props.serverUrl}</p>
    </>
}

export default PaymentsList;

export type PaymentsListProps = {
    serverUrl : string
}
```

- Change the indexPaymentsList.tsx file to accept properties from the container as a parameter to the mount function, and pass these to the component

```
import {createRoot} from 'react-dom/client';
import './../index.css';
import PaymentsList, { PaymentsListProps } from './PaymentsList';

const mount = (el: Element, props: PaymentsListProps) : void => {

  const root  = createRoot(el);

  root.render(<PaymentsList {...props} />);
```

```
}

export { mount };
```

- Change the App.tsx file to include properties when run locally

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import PaymentsList from './PaymentsList/PaymentsList';

function App() {
  return (
    <div>
      <h1>Payments list app</h1>
      <PaymentsList serverUrl='http://someServer'/>
    </div>
  );
}

export default App;
```

Now in the container app payments-ui – remotecomponents/PaymentsList,

- Pass properties to the mount function

```
import React, { useEffect, useRef } from 'react';
import {mount} from 'paymentslist/PaymentsList';

const PaymentsList = () : JSX.Element => {

  const paymentslistdiv = useRef<HTMLDivElement>(null);

  useEffect( () => {
    mount(paymentslistdiv.current, {serverUrl:
"https://payments.multicode.uk/"});
  } , []);
```

```
  return (
    <div ref={paymentslistdiv} />
  );
}

export default PaymentsList;
```

Test this is working, then suggest that the students:

- replicate the same thing for payments-add.

- for now don't use stateful variables.

2. Communication between Container and Payment-add app

Follow similar steps as above to pass the url to payment-add app.

3.Activity 2 – Create a context in container and pass to remote mfe

In this activity, we will practice how we can pass change of state between container and micro-app using functions.

Pre-requisites

You should have a running "payments-ui" MFE application running

Challenge requirements

1. Create a context in the container to store the state of the current user – whether they are logged in or not.

2. Use the state of the user to determine whether or not to show the menu (pass the state to the relevant component)

3. In the home page, create a login / logout button. The button should tell the container that the user has logged in / logged out

4. Do not try and update the state of the child component

1. Changes to HomePage micro-app component

1. Edit the home page component to take properties: (1) initial user state, (2) functions to call to mark a user as logged in or logged out. Change HomePage, indexHomePage and App

static-pages/HomePage.tsx

```
type user = {id:number, name:string, role:string};

export type HomePageProps = {id:number, login:(user:user)=>void, logout:()=>void};


const HomePage = (props:HomePageProps) => {


   return (
      <h1>This is the home page mfe</h1>
   )
}

export default HomePage;
```

*/static-pages/indexHomePage.tsx

```
import  ReactDOM from 'react-dom/client';
import HomePage, { HomePageProps } from './HomePage';

const mount = (el:Element, props:HomePageProps):void => {

   const root:ReactDOM.Root = ReactDOM.createRoot(el);
   root.render(<HomePage {...props} />)

}

export {mount};
```

static-pages/App.tsx

```
import React from 'react';
import logo from './logo.svg';
import './App.css';
import HomePage from './HomePage/HomePage';
import PageNotFound from './PageNotFound/PageNotFound';

function App() {
  return (
```

```
    <div>
    <h1>Home page</h1>
    <HomePage id={0} login = { ()=> {} } logout = { () => {} } />
    <PageNotFound />
    </div>
  );
}

export default App;
```

2. Store the login state locally in the HomePage component, because it will not get notified of changes in state

static-pages/HomePage.tsx

```
type user = {id:number, name:string, role:string};

export type HomePageProps = {id:number, login:(user:user)=>void, logout:()=>void};


const HomePage = (props:HomePageProps) => {

    const loginState = props.id!==0;
    return (
        <h1>This is the home page mfe</h1>
    )
}

export default HomePage;
```

3. Create a context object in the container (payments-ui) to store the logged in user. Create a folder called "context" under src folder. Create a file called context.tsx. This is same as what was defined in standlone payments-ui app developed during react.

context.tsx

```
import { createContext } from "react";
```

```
export type userType = {id:number, name:string, role:string};
export type userContextType = userType & {login:(user:userType)=> void, logout:()=>void};

export const UserContext:React.Context<userContextType> = createContext<userContextType>({id:0,
name:"", role:"", login:()=>{}, logout:()=>{}})
```

4. Edit the container (payments-ui) to pass the relevant properties to remote home page component

App.tsx

```
import React, { useEffect, useRef, useState } from 'react';
import './App.css';
// import {mount} from 'paymentslist/PaymentsList';
import PaymentsList from './remoteComponents/PaymentsList';
import PaymentsAdd from './remoteComponents/PaymentsAdd';
import HomePage from './remoteComponents/HomePage';
import PageNotFound from './remoteComponents/PageNotFound';
import { BrowserRouter, Route, Routes } from 'react-router-dom';
import PageHeader from './components/PageHeader/PageHeader';
import { UserContext, userType } from './context/context';

function App() {

 const [user,setUser] = useState<userType>({id:0, name:'', role:''});

 const login = (user:userType) => {
  setUser(user);
 }

 const logout = () => {
  setUser({id:0, name:'', role:''});
 }


 return (
  <UserContext.Provider value={{...user, login:login, logout:logout}}>
  <BrowserRouter>
```

```
    <PageHeader />
    <Routes>
      <Route path="/find" element={<PaymentsList />} />
      <Route path="/add" element={<PaymentsAdd />} />
      <Route path="/" element={<HomePage />} />
      <Route path="*" element={<PageNotFound />} />
    </Routes>
  </BrowserRouter>
  </UserContext.Provider>
 );
}

export default App;
```

remoteComponents/HomePage

```
import {mount} from 'staticpages/HomePage';
import { useContext, useEffect, useRef } from 'react';
import { UserContext, userContextType, userType } from '../context/context';

const HomePage = ():JSX.Element => {

  const userContext = useContext<userContextType>(UserContext);

  const homepagediv = useRef<HTMLDivElement>(null);
  useEffect(()=> {
    mount(homepagediv.current, {id:userContext.id, login:login, logout:logout});
  },[]);

  const login = (user:userType) => {
    userContext.login(user);
  }

  const logout = () => {
    userContext.logout();
  }

  return(
    <div ref={homepagediv}/>
  )
}
```

```
export default HomePage;
```

5. Edit the PageHeader componet to display the menu based on the login state in the container (payments-ui) app. i.e. if id is 0, then dont display the menu

PageHeader.tsx

```
import './pageHeader.css';
import Menu from "./Menu";
import { Link } from 'react-router-dom';
import { useContext } from 'react';
import { UserContext, userContextType } from '../../context/context';

const PageHeader = () : JSX.Element => {

   const userContext = useContext<userContextType>(UserContext);


   return (
      <div className="pageHeader">
         <h1><Link to="/">Payments Application</Link></h1>
         {userContext.id !== 0 && <Menu/>}
      </div>
   );
}

export default PageHeader
```

6. Edit the micro-app Homepage component to display the id received in props.

static-pages/HomePage.tsx

```
type user = {id:number, name:string, role:string};

export type HomePageProps = {id:number, login:(user:user)=>void, logout:()=>void};
```

```
const HomePage = (props:HomePageProps) => {

   const loginState = props.id!==0;
   return (
      <>
      <h1>This is the home page mfe</h1>
      <p>login id = {props.id}</p>
      </>
   )
}

export default HomePage;
```

7. Edit the micro-app homepage component to display button login or logout based on the loginState it maintains. Also
invoke the props.login function with a dummy user. The login button - invokes the login function passed by the container - which sets id to 1.
This re-renders the PageHeader component to display the menu items.
But the id is still shown 0 in the homepage micro-app which we will do in the next step.

static-pages/HomePage

```
type user = {id:number, name:string, role:string};

export type HomePageProps = {id:number, login:(user:user)=>void, logout:()=>void};


const HomePage = (props:HomePageProps) => {

   const loginState = props.id!==0;

   const login = () => {
      props.login({id:1, name : "Matt", role : "manager"})
   }


   const logout = () => {
      props.logout();
   }
```

```
  return (
    <>
    <h1>This is the home page mfe</h1>
    <p>{props.id}</p>
    {!loginState && <button onClick={login} >Login</button>}
    {loginState && <button onClick={logout}>Logout</button>}

    </>
  )
}

export default HomePage;
```

4.Using functions for state change

- In the remote component, we have the mount function. So far we have had this as a function which takes parameters, but doesn't return anything – it's a void function.
- We have called this from the container, passing in the element and any properties that we want to pass to the remote component
- Now we are going to say that the mount function we declare does return something. It's going to return an object of key value pairs, where the keys are names and the values are functions.
- This means that when we call the mount function we'll get back an object containing the function that was defined in the remote component
- Now the container object has a function it can call, which will be executed in the remote object.

To implement this…

In indexHomePage, we're going to change our mount function so that it returns a function that the container can call.

```
import ReactDom, { Root } from 'react-dom/client';
import './../index.css';
import HomePage, { HomePageProps } from './HomePage';

type mountReturnType = { onLoginStateChanged : (newId: number) => void};

const mount = (el: Element, props: HomePageProps) : mountReturnType => {

  const root = ReactDom.createRoot(el);
```

```
  root.render(<HomePage {...props} />);

  const parentLoginStateChanged = (newId : number) => {
    root.render(<HomePage {...props} id = {newId} />);
  }

  return {onLoginStateChanged : parentLoginStateChanged}
}

export { mount };
```

Now when we call mount, we'll get back a reference to a function called onLoginStateChanged. This function when it is executed is going to re-render the homepage with new properties. Note that we are re-using "root" we are not re-creating it so we won't see that error in the console we saw before.

We need to store that function when we receive it somewhere so let's add it to the context object in the container

```
import {createContext} from "react";

export type userType = {id: number, name : String, role : String};
export type homepageMountReturnType = { onLoginStateChanged : (newId: number) =>
void};

export type userContextType = userType & homepageMountReturnType & { login :
(user : userType) => void, logout : () => void};

export const UserContext : React.Context<userContextType> =
    createContext<userContextType>({id: 0, name : "", role : "", login : () =>
{}, logout: () => {}, onLoginStateChanged : (newId: number) => {} });
```

Now in in the container's app component, where we provide the context we need to put a value in for the function that can later be changed

```
    <UserContext.Provider value={{...user, login : login, logout : logout,
onLoginStateChanged : () => {} }}>
```

App.tsx now looks like this:

```
import React, { useState } from 'react';
import logo from './logo.svg';
import './App.css';
import PaymentsAdd from './remoteComponents/PaymentsAdd';
import PaymentsList from './remoteComponents/PaymentsList';
import { BrowserRouter, Route, Routes } from 'react-router-dom';
import PageHeader from './components/pageHeader/PageHeader';
import HomePage from './remoteComponents/HomePage';
import PageNotFound from './remoteComponents/PageNotFound';
import { UserContext, userType } from './context/context';
```

```
function App() {

const [user,setUser] = useState<userType>( {id: 0, name : "", role : ""});



 const logout = () => {
      setUser({id: 0, name : "", role : ""});
  }

  const login = (user : userType) => {
      setUser(user);
  }

  return (
    <UserContext.Provider value={{...user, login : login, logout : logout,
onLoginStateChanged : () => {} }}>
    <BrowserRouter>
      <PageHeader />
      <Routes>
        <Route path="/find" element={<PaymentsList />} />
        <Route path="/add" element={<PaymentsAdd />} />
        <Route path="/" element={<HomePage />} />
        <Route path="*" element={<PageNotFound />} />
      </Routes>
    </BrowserRouter>
    </UserContext.Provider>
  );
}

export default App;
```

and in the container's homepage component we can set the function:

```
  useEffect( () => {
    const result = mount(homepagediv.current, {...userContext});
    console.log("result",result == null);
    userContext.onLoginStateChanged = result;
  } , []);
```

To make this work at the moment we are just passing the userContext to the homepage function when we mount it… we need to do something a little different now:

```
import React, { useContext, useEffect, useRef, useState } from 'react';
import {mount} from 'staticpages/HomePage';
import { UserContext, homepageMountReturnType, userContextType, userType } from
'../context/context';

const HomePage = () : JSX.Element => {
```

```
  const userContext = useContext<userContextType>(UserContext);
  const homepagediv = useRef<HTMLDivElement>(null);

  const login = (user: userType) : void => {
    userContext.login(user);
    userContext.onLoginStateChanged(user.id);
  }

  const logout = () : void => {
    userContext.logout();
    userContext.onLoginStateChanged(0);
  }

  useEffect( () => {
    const result : homepageMountReturnType = mount(homepagediv.current, {id :
userContext.id, login, logout});
    userContext.onLoginStateChanged = result.onLoginStateChanged;
  } , []); //note no dependencies so that we don't update the child component
when the user state changes

  return (
    <div ref={homepagediv} />
  );
}

export default HomePage;
```

These login and logout functions are a bit odd… taking loging as an example, we pass this function to the HomePage remote component as a property. In HomePage when the user clicks on login this function is called. That then (1) changes the value in our context object, and (2) passes the value back to the remote component which is then re-rendered.
Finally back to the static pages HomePage component, we can remove the stateful variable as we are now being notified when the value changes:

```
const HomePage = (props: HomePageProps) : JSX.Element => {

    const loginState = props.id !== 0;

    const login = () => {
        props.login({id:1, name : "Matt", role : "manager"})
    }

    const logout = () => {
        props.logout();
    }

    return (<div>
        <h3>Welcome to the Payments application.</h3>
```

```
        {!loginState && <button onClick={login} >Login</button>}
        {loginState && <button onClick={logout}>Logout</button>}
    </div>);
}

export default HomePage;

type user = {id:number, name: string, role: string};

export type HomePageProps = {id : number, login: (user: user) => void, logout :
() => void }
```