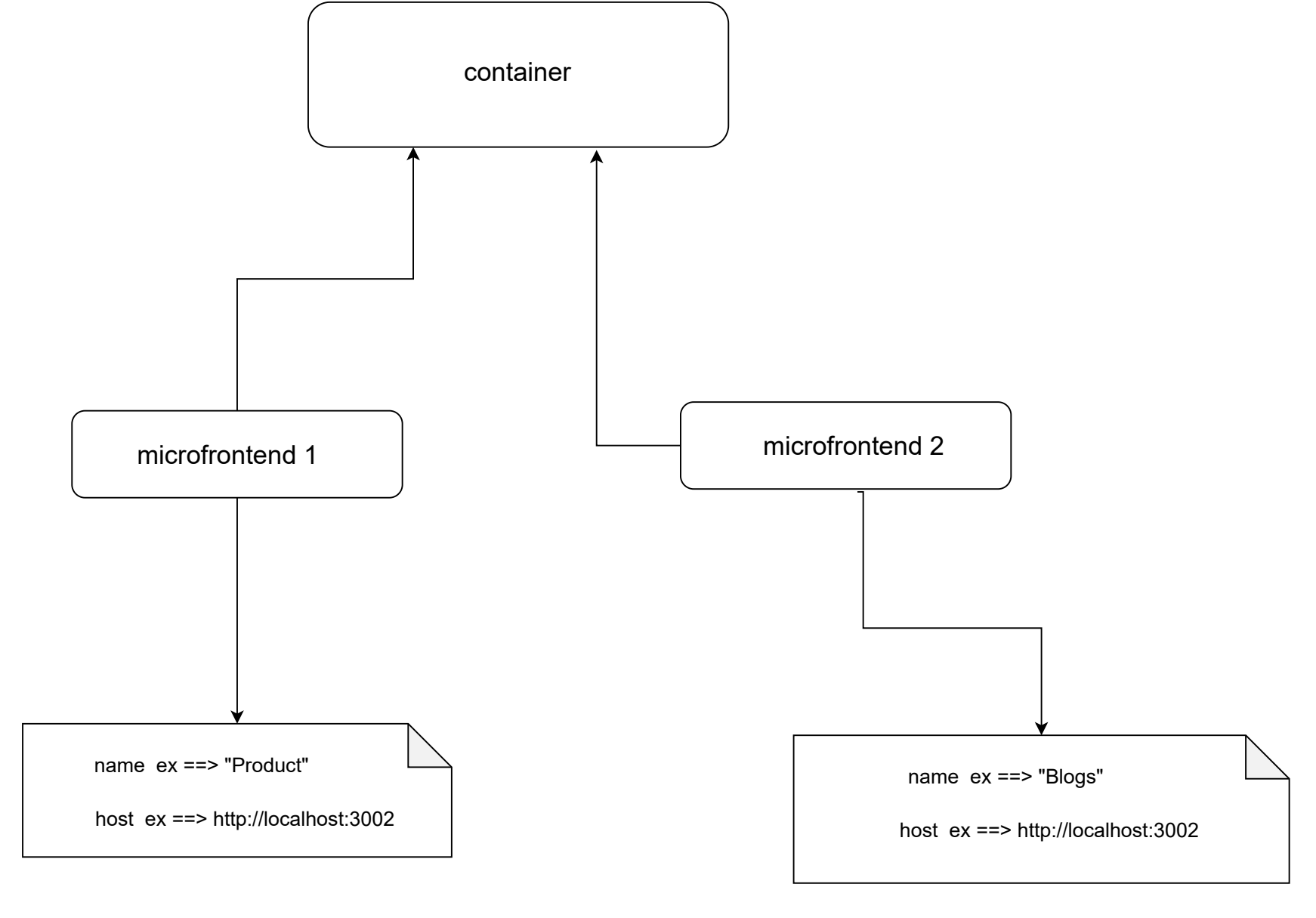
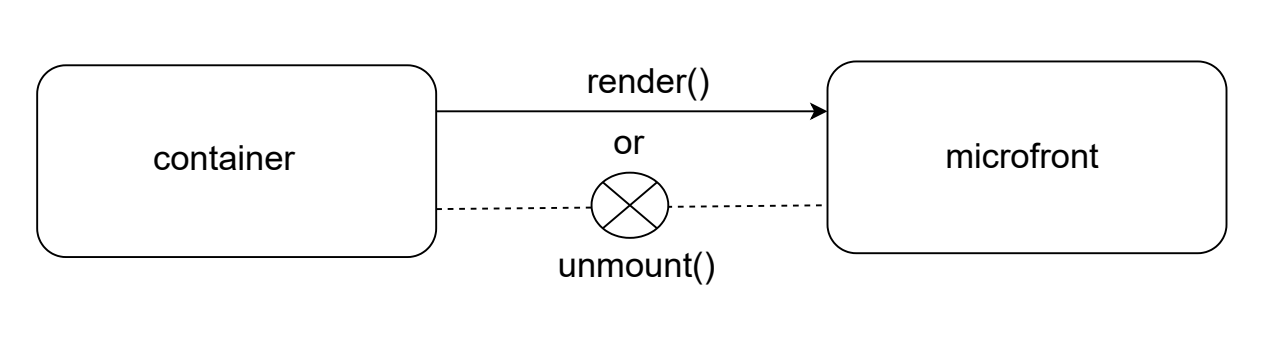
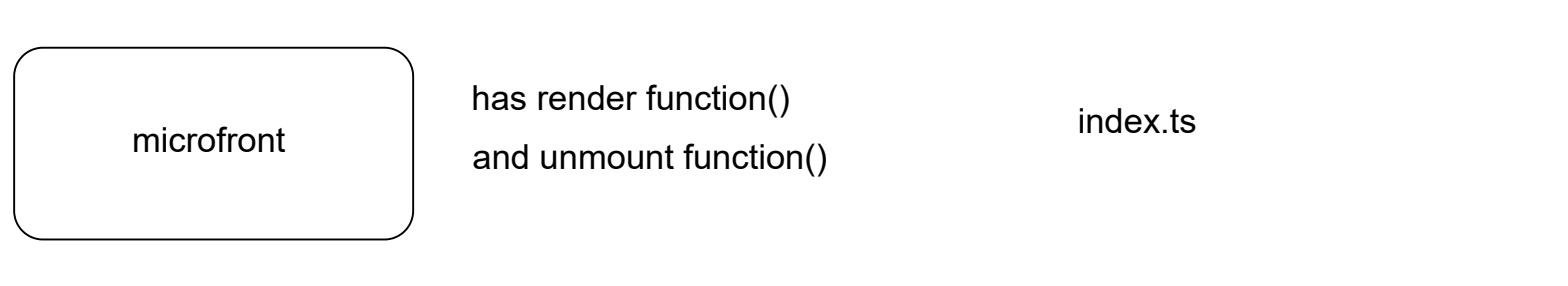
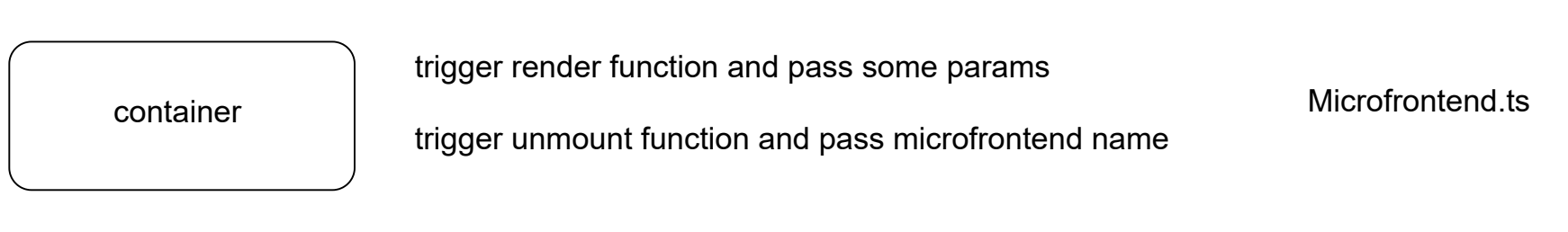






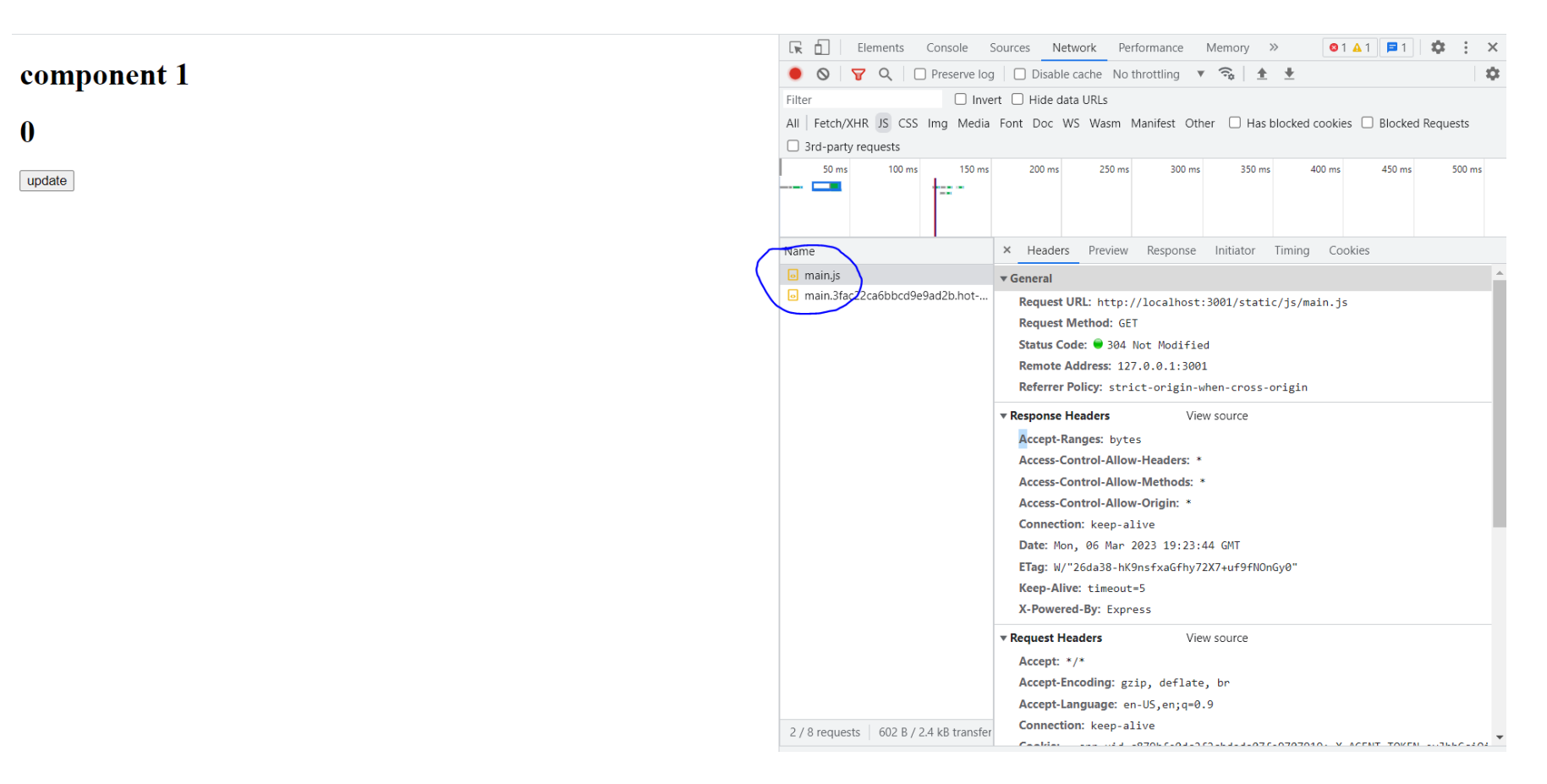


Rendering MicroFrontend

how container render microfrontend

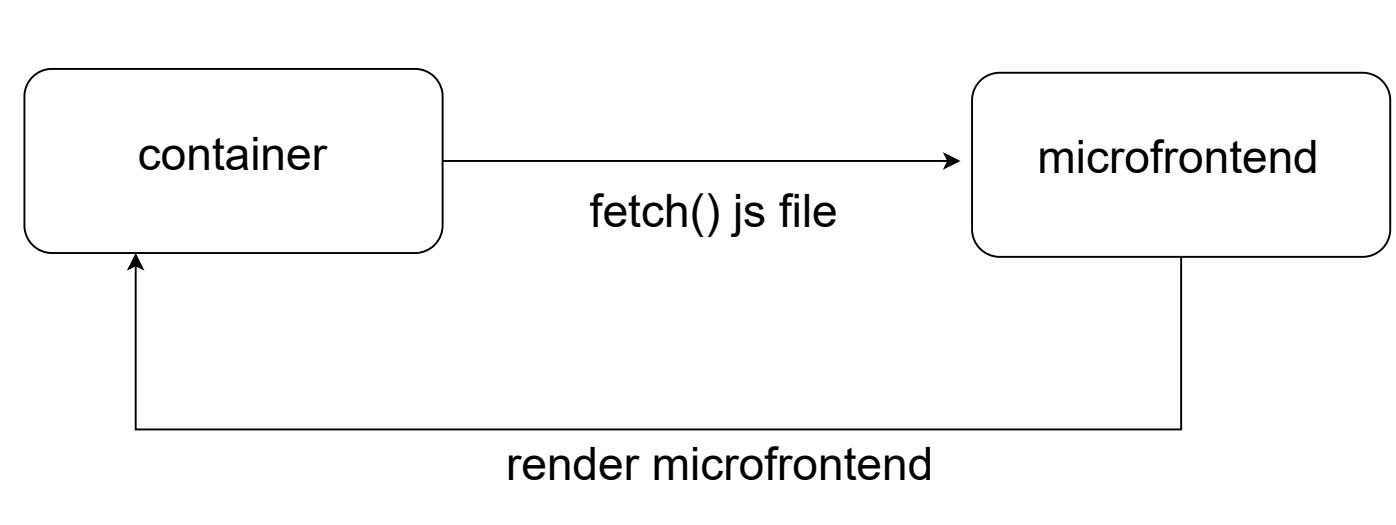
react need to generate a bundle js file contain some js code to make browser know about your application to do that react use webpack as module bundler

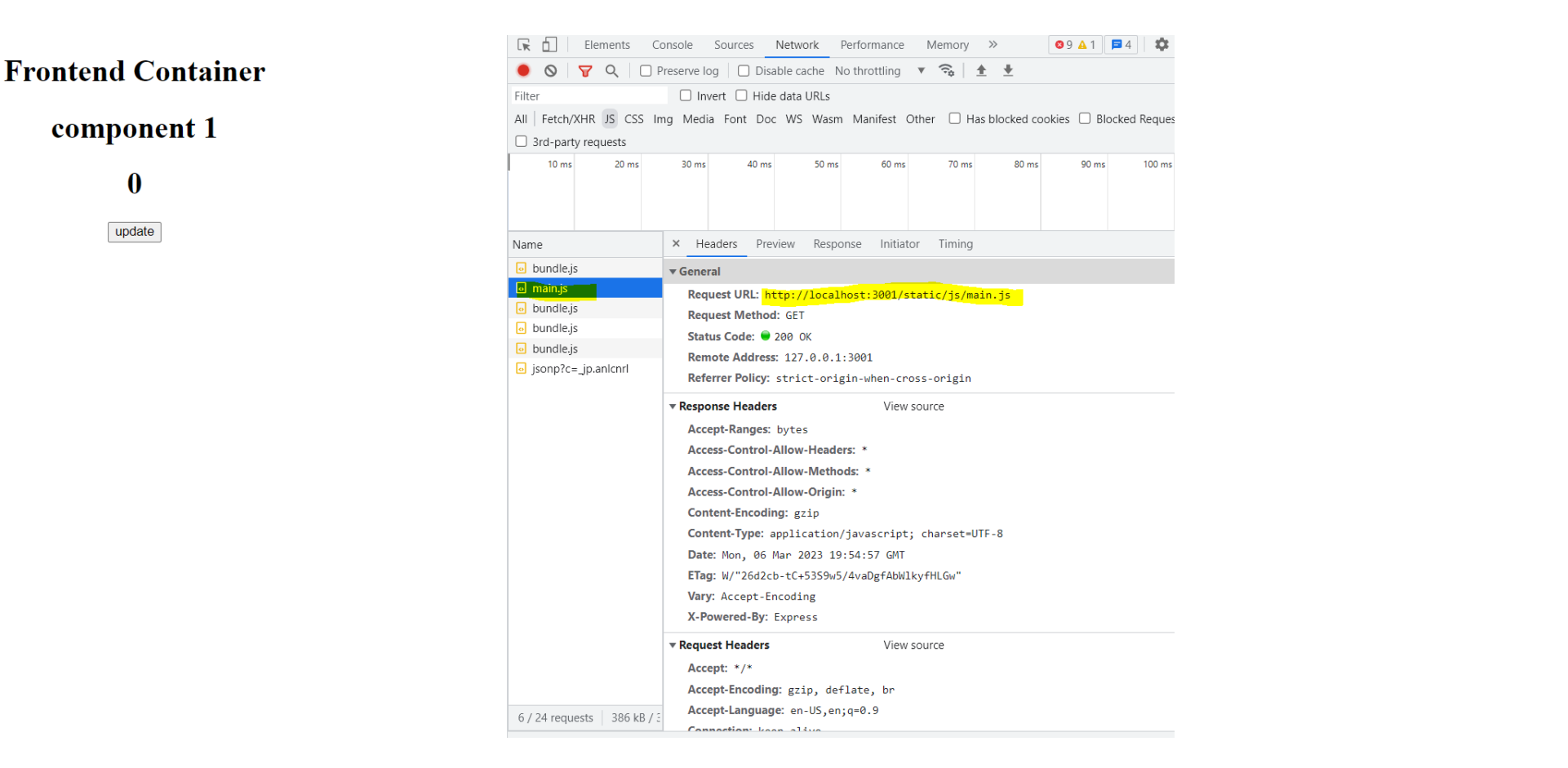
webpack generate some assets files and the most important one is js file



main.js contain all javascript code generated by webpack it's actually your react code

now this information it's very important by frontend container because we want to do a fetch request to this file





now let's setup our first microfrontend

requirements

install node version 14.16.0

npm config set proxy http://proxyrma.rmawatanya.com:8080

install yarn using npm : npm install `--global yarn`

```
npx create-react-app products
```

```
yarn add react-router-dom@5.3.4
```

```
yarn add history@5.3.0
```

crearte your first component

```
import React from 'react'

const Products = () => {

  return (
    <div>
      <h1>component 1</h1>
      <h1>0</h1>
      <button>update</button>
    </div>
  )
}

export default Products
```

make a route for it

```
const defaultHistory = createBrowserHistory();
function App({ history = defaultHistory }) {
  return (
    <div className="App">
    <Router>
    <Switch>
     <Route exact path="/products" component={Products}/>
    </Switch>
    </Router>
    </div>
  );
}

export default App;
```

Container Application

```
npx create-react-app container
```

```
yarn add react-router-dom@5.3.4
```

or

```
npm i react-router-dom@5.3.4
```

```
yarn add history@5.3.0
```

Create Microfrontend Component

```
import React, { useEffect } from "react";

function MicroFrontend({ name, host, history }) {
 useEffect(() => {
   const scriptId = `micro-frontend-script-${name}`;
    console.log(scriptId);
   const renderMicroFrontend = () => {

     window[`render${name}`](`${name}-container`, history);
     console.log(`microfrontend ${name} load successfully`);

   };

   if (document.getElementById(scriptId)) {
     renderMicroFrontend();
     return;
   }

   fetch(`${host}/asset-manifest.json`)
     .then((res) => res.json())
     .then((manifest) => {
      console.log("manifest",manifest);
       const script = document.createElement("script");
       script.id = scriptId;
       script.crossOrigin = "";
       console.log(manifest);
       script.src = `${host}${manifest['main.js']}`;
       script.onload = renderMicroFrontend
       document.head.appendChild(script);
     });

   return () => {
     window[`unmount${name}`] && window[`unmount${name}`](`${name}-container`);
   };
 });

 return <main id={`${name}-container`} />;
}

MicroFrontend.defaultProps = {
 document,
 window,
};

export default MicroFrontend;
```

What happen above !!!!!

This code defines a React component called MicroFrontend that is responsible for rendering a micro frontend inside a container with a given name. A micro frontend is a small, self-contained part of a larger web application that is typically developed by a separate team and can be deployed independently.

The MicroFrontend component takes three props: name, host, and history. The name prop is a unique identifier for the micro frontend, the host prop is the URL of the server where the micro frontend is hosted, and the history prop is the browser history object that is passed down from the parent application.

Inside the MicroFrontend component, the useEffect hook is used to manage the lifecycle of the micro frontend. This hook is called after the component has mounted and every time any of its dependencies change.

The useEffect hook first generates a unique ID for the script element that will be added to the page to load the micro frontend. If a script with the same ID already exists on the page, the micro frontend is rendered immediately, and the useEffect hook returns early.

If a script with the specified ID does not exist on the page, the code fetches the asset-manifest.json file from the server where the micro frontend is hosted. This file contains information about the files that make up the micro frontend, including the name of the main JavaScript file.

The code then creates a new script element, sets its ID and src attributes to the appropriate values, and adds it to the head of the document. When the script has finished loading, the renderMicroFrontend function is called to render the micro frontend inside the container with the given name.

The useEffect hook also returns a cleanup function that is responsible for unmounting the micro frontend when the component is unmounted. This is achieved by calling the unmount function that is exported by the micro frontend and passing in the name of the container where it is mounted.

Finally, the MicroFrontend component returns a main element with the same ID as the container where the micro frontend will be rendered. This element serves as a placeholder until the micro frontend is loaded and rendered inside it.

create simple microfrontends file and call our custom microfrontend component

first create .env vile to define your microfrontends hosts

```
REACT_APP_PRODUCTS_HOST=http://localhost:3001
```

in microfrontends.js

```
import MicroFrontend from "./components/Microfrontend/MicroFrontend";
const {
    REACT_APP_PRODUCTS_HOST:ProductsHost,
   } = process.env;

export function Products({ history }) {
    return <MicroFrontend history={history} host={ProductsHost} name="Products"
/>;
   }
```

passing the host , history , name to the microfrontend

now add your microfrontend to the route

```
import './App.css';
import { createBrowserHistory } from "history";
import {Switch , Route , BrowserRouter as  Router} from "react-router-
dom"
import React from 'react';
import { Blogs, Products } from './microfrontends';
const defaultHistory = createBrowserHistory();


function App({history=defaultHistory}) {
  return (
    <div className="App">
    <h1>Frontend Container</h1>
    <Router>
    <Switch>
    <Route exact path="/products" component={Products}/>
    </Switch>
    </Router>
    </div>
  );
}

export default App;
```

add some config for Products Application

```
yarn add react-app-rewired
```

→ to work with overrid.js file

Create config.overrides.js in the root directory of blogs application and add the following code.

```
module.exports = {
    webpack: (config, env) => {
      config.optimization.runtimeChunk = false;
      config.optimization.splitChunks = {
        cacheGroups: {
          default: false,
        },
      };
       config.output.filename = "static/js/[name].js";
       config.plugins[5].options.filename = "static/css/[name].css";
      config.plugins[5].options.moduleFilename = () =>
"static/css/main.css";
      return config;
    },
    devServer: function(configFunction) {
      // Return the replacement function for create-react-app to use
to generate the Webpack
      // Development Server config. "configFunction" is the function
that would normally have
      // been used to generate the Webpack Development server config -
you can use it to create
      // a starting configuration to then modify instead of having to
create a config from scratch.
      return function(proxy, allowedHost) {
        // Create the default config by calling configFunction with
the proxy/allowedHost parameters
        const defaultConfig = configFunction(proxy, allowedHost);
        return {
          ...defaultConfig,
          hot: true,
          historyApiFallback: true,
          // Allow mounting react apps from a page served by the
backend.
          headers: {
            'Access-Control-Allow-Origin': '*',
            'Access-Control-Allow-Headers': '*',
            'Access-Control-Allow-Methods': '*',
          },
        };
      };
    },


  };
```

Now, let's update scripts section of package.json file,

```
  "scripts": {
    "start": "set PORT=3001 && react-app-rewired start",
    "build": "react-app-rewired build",
    "test": "react-app-rewired test",
    "eject": "react-app-rewired eject"
  },
```

And final step in Products Application to update index.js,

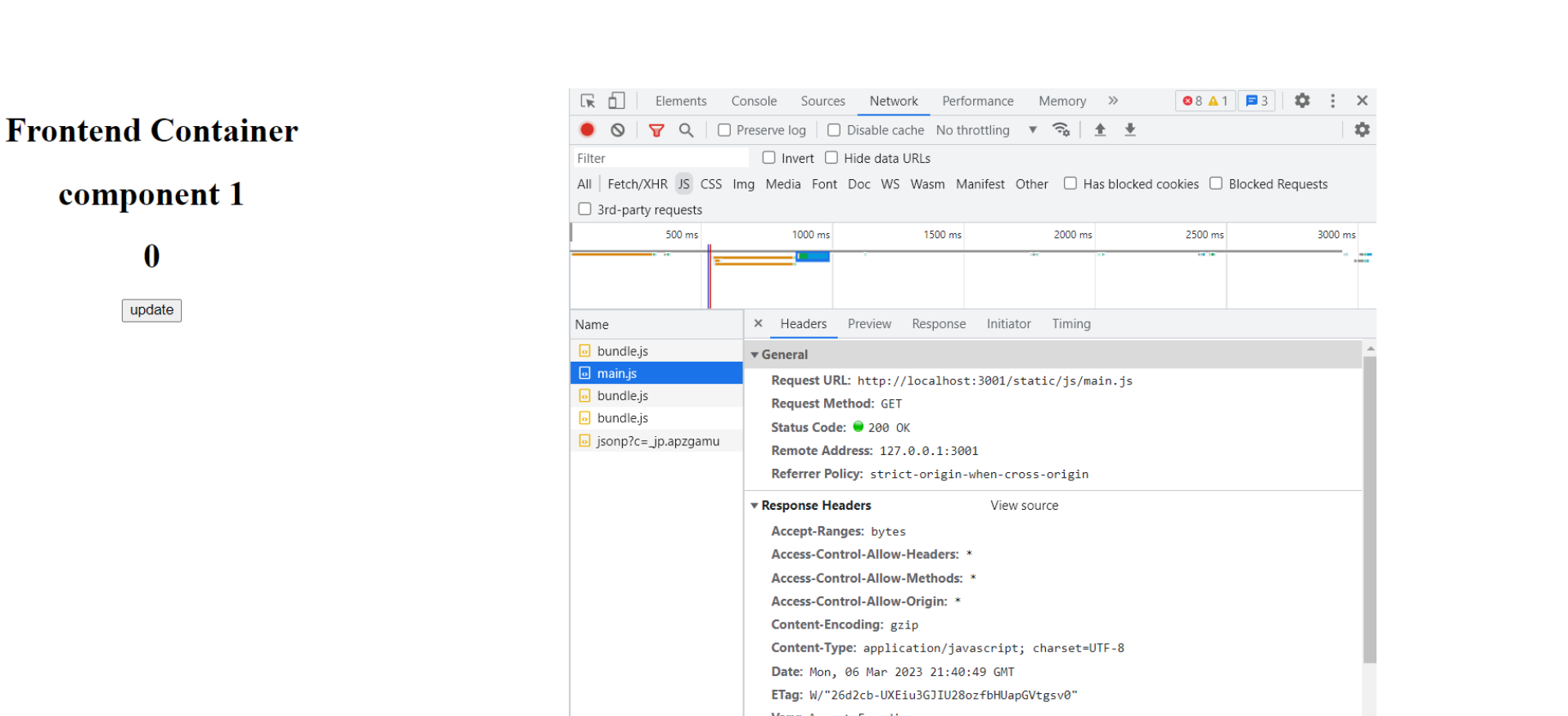
```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

window.renderProducts = (containerId, history) => {
 ReactDOM.render(
   <App history={history} />
   document.getElementById(containerId),
 );
};

window.unmountProducts = containerId => {
 ReactDOM.unmountComponentAtNode(document.getElementById(containerId));
};

if (!document.getElementById('Products-container')) {
 ReactDOM.render(
   <App />
  , document.getElementById('root'));
}
```

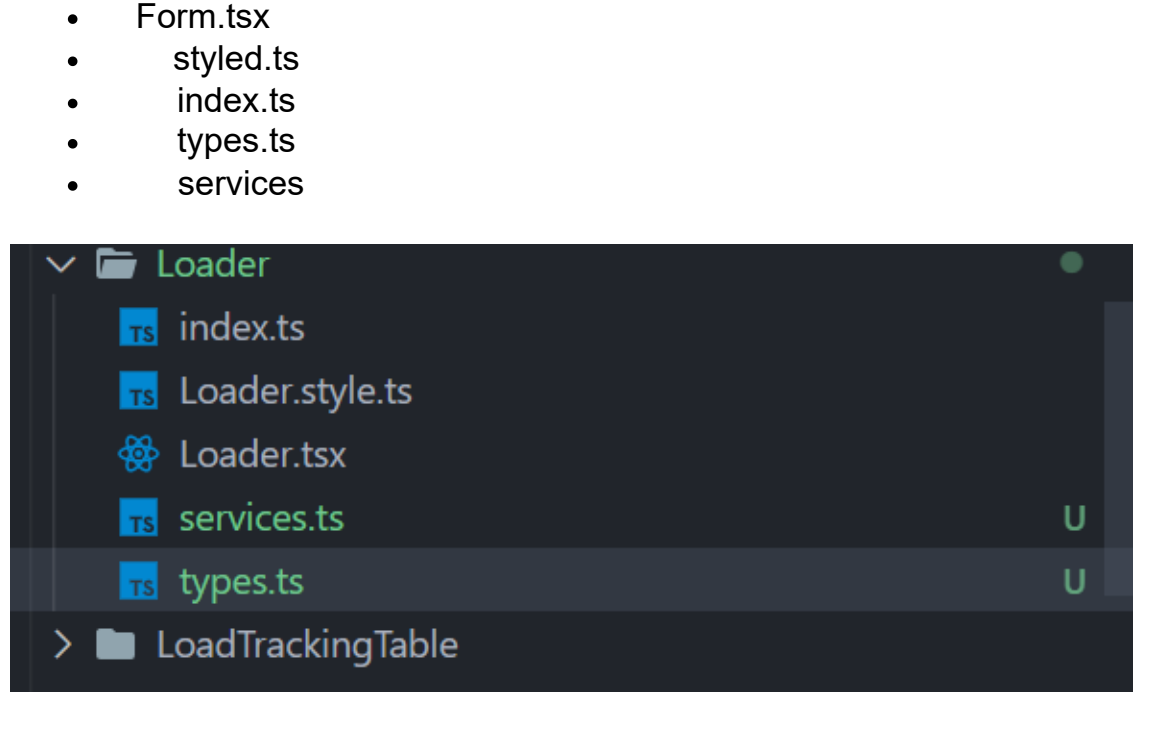
Boooom Congratulation !!!!!!!!!!!!!!!!!!



component creation best practice

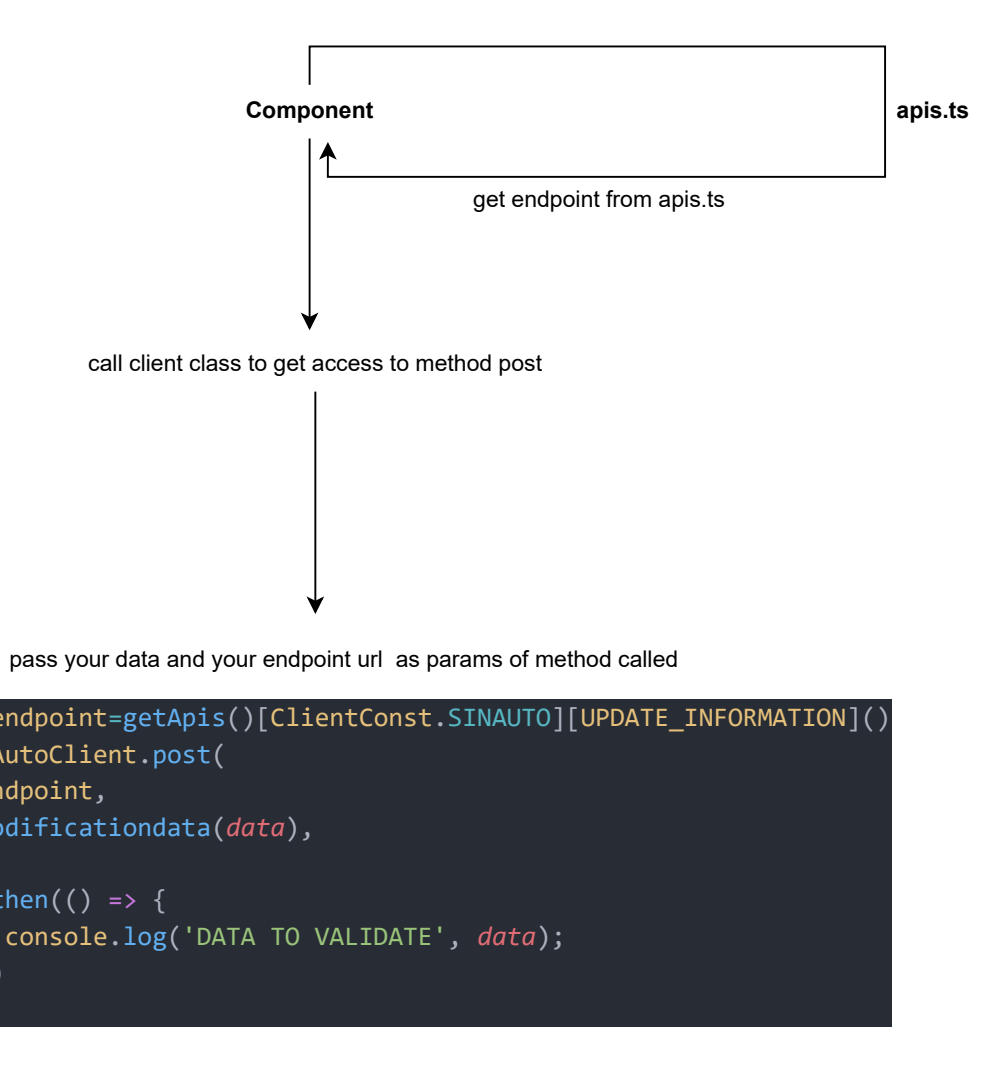
Components
Form

- Form.tsx
- styled.ts
- index.ts
- types.ts
- services



Api Call Flow

from component directly




```
[ClientConst.SINAUTO]: {
  [UPDATE_INFORMATION]: () => '/predeclaration/api/predeclaration/modifier',
},
```

```
const endpoint=getApis()[ClientConst.SINAUTO][UPDATE_INFORMATION]()
   SinAutoClient.post(
     endpoint,
     modificationdata(data),
   )
     .then(() => {
       console.log('DATA TO VALIDATE', data);
     })
 };
```

client class post method

```
 post(endpoint: string, data: object, authorizedStatus: number[] =
[]) {
   return trackPromise(
     this.request('post', endpoint, data,
authorizedStatus).then(response => {
       return response.body;
     }),
   );
 }
```