# Frontend con React y redux

guia con ejemplos para construir un cliente con react.js que manejará y mostrará datos obtenidos de una API de express.

## para iniciar:

> en la carpeta **raiz** de nuestro proyecto (es decir, la carpeta que contiene la carpeta del backend):

anteriormente usabamos el comando create-react-app *(CRA)*,
ahora tenemos la opción de usar vite, que agiliza procesos y tiene beneficios en velocidad, optimizacion de memoria y errores detallados, asi que:

> **npm create vite@latest**
>
> - React.
> - Javascript
> - NombreDeMiProyecto
>   con esto lograremos que dentro de la carpeta raiz del proyecto, tengamos la carpeta raiz de la API, y la carpeta raiz del Cliente.

con esto tendremos una estructura similar a:

> /carpeta global del proyecto
>
> > /carpeta del backend
> >
> > /carpeta del frontend

de aqui en adelante, consideraremos como *Raíz* a la "carpeta del frontend"
que lleva el nombre que definimos en el menu de create vite.

dentro de nuestra carpeta raiz del frontend ejecutaremos:

```
npm i redux@4.0.5 redux-thunk react-redux react-router-dom@5.2.0 axios
```

cuando usabamos *CRA* ejecutabamos el comando *npm start* para correr nuestra app en el puerto *3000*.
En Vite, se usa el comando *npm run dev*
y el proyecto se levanta en el puerto *5173*.

En este punto podriamos "limpiar nuestro proyecto" de algunas dependencias que no son *del todo* necesarias o que podrian confundirnos si no las conocemos.

"Limpiar" implica eliminar archivos, dependencias e importaciones de estas dependencias y archivos que no usaremos.

Eliminaremos todo lo referente a:

- setupTests
- reportWebVitals
- logos de react y vite
- index.css, app.css, main.css
- app.test.js
- en app.js index.js o main.js eliminar todos los imports y llamadas que refieran a lo que hemos eliminado
- en app.jsx eliminaremos todo lo que esté dentro del return principal asi como la definicion del estado local *count* que viene por defecto quedando nuestro App.jsx asi:

```
function App() {
    return <main></main>;
}
export default App;
```

podriamos agregar un titulo H1 dentro de nuestra etiqueta principal psimplemente para ver, y verificar que todo anda bien

En index.html podemos editar el titulo y el icono de la pestaña del navegador.

## estructura

organizaremos los componentes en carpetas según su funcion, recordando el concepto de dumb components y de smart components podemos determinar que algunos componentes de nuestra app, serán "vistas" y otros componentes serán "componentes" de cada vista.
por ejemplo.
las vistas son: landing, login, home, detail, etc...
los componentes son: navBar, Footer, card, etc...

Con esto en mente, nuestra estructura será:

> /
> /index.html
> /package.json
> /.gitignore
>
> > /public
>
> > /src
> > /src/app.jsx
> > /src/main.jsx
> >
> > > /src/assets

> /src/views
> /src/views/landing.jsx
> /src/views/home.jsx
> /src/views/detail.jsx

> /src/components
> /src/components/navBar.jsx
> /src/components/card.jsx
> /src/components/Footer.jsx

## Rutas

Si bien ya podemos crear componentes que se muestren uno dentro de otro y asi definir vistas, *hasta este momento* estariamos viendo todas nuestras vistas y sus componentes dentro del componente principal "App.jsx" y en la url *http://localhost:5173***/** (para fines prácticos esto es simplemente '**/**').

para lograr que cada vista se vea en una URL distinta como por ejemplo

- landing en **/**
- home en **/home**
- el detalle de un usuario en **/detail/5**

haremos uso de un router, en nuestro caso, *react-router-dom*, cuya implementación es la siguiente:

1. en **index.js** importaremos

```
import { BrowserRouter } from "react-router-dom";
```

y anidaremos nuestra etiqueta dentro de la etiqueta BrowserRouter

```
<BrowserRouter>
    <App />
</BrowserRouter>
```

obteniendo la capacidad de manejar rutas para toda nuesta aplicación podemos porceder a definir las rutas en App.jsx de las siguientes 3 formas:

```
import Home from "./views/Home/Home.jsx";
import Landing from "./views/Landing/Landing.jsx";
import Detail from "./views/Detail/Detail.jsx";
import { Route } from "react-router-dom";
function App() {
    return (
```

```
        <main className="App">
            <Route
                exact
                path="/">
                <Landing />
            </Route>
            /* permite pasarle props*/
            <Route
                path="/home"
                component={Home}
            />
            /* no permite pasarle props */
            <Route
                path="/detail"
                render={() => <Detail />}
            />
            /* permite pasarle props */
        </main>
    );
}
export default App;
```

> solo se usa exact path para la ruta raiz, ya que todas las rutas comienzan con **'/'** y todas mostrarían el contenido del componente landing si no usamos exact path

podemos optimizar las importaciones de cada modulo, implementando el uso de un index en la carpeta views, el cual va a importar y exportar todas las views, de modo que en otros modulos no sea necesario especificar la ruta completa de cada importación, sino que importaremos la view que necesitemos directamente desde la carpeta views, gracias a nuestro index.

> '/src/views/index.js'
>
> ```
> import Home from "./views/Home/Home.jsx";
> import Landing from "./views/Landing/Landing.jsx";
> import Detail from "./views/Detail/Detail.jsx";
> import Form from "./views/Form/Form.jsx";
>
> export { Detail, Form, Home, Landing };
> ```

Y asi podriamos reemplazar todas las importaciones en App.jsx por una única importacion múltiple:

```
import { Detail, Form, Home, Landing } from "./views";
```

Esta implementacion de index, no hace gran diferencia cuando tenemos tres importaciones en App, pero sí cuando tenemos 15 o mas, así que es un dato util de conocer y dominar, yque podemos usar en cualquier carpeta que contenga varios modulos, por ejemplo en components.

> es común que existan componentes que querramos que se muestren por sobre los demás componentes, ó sobre algunos, ó sobre todos excepto algunos, como es el caso de la navBar, que normalmente en la landing page no queremos verla, pero en los demas componentes si.
> y si bien, podemos importarla en cada uno de los componentes que deseamos que se muestre, tambien podriamos mostrarla por fuera de las rutas, asi sería visible sin importar la url.
> pero ¿que pasa con landing? ¿cómo logro que no se vea la NavBar en Landing?

una opcion, es valernos del hook useLocation, y determinar segun la url, un renderizado condicional del componente NavBar desde App.jsx:

```
import { Detail, Form, Home, Landing } from "./views";
import { NavBar } from "./components";
import { Route, useLocation } from "react-router-dom";

function App() {
    const location = useLocation();
    const pathName = location.pathname;

    return (
        <main className="App">
            {pathName !== "/" && <NavBar />}
            /* asi podemos usar la lógica de javascript para decidir que el
componente
            NavBar se muestre solo cuando la ruta sea distinta de "/" */
            <Route
                exact
                path="/"
                render={() => <Landing />}
            />
            <Route
                path="/home"
                render={() => <Home />}
            />
            <Route
                path="/create"
                render={() => <Form />}
            />
            <Route
                path="/detail"
                render={() => <Detail />}
            />
        </main>
    );
```

```
}
export default App;
```

## componentes

cada modulo o archivo jsx en nuestra app de react, es un componente. los componentes pueden construirse como una función ( llamados componentes funcionales ) o mediante una clase que esxtiende de react, (componentes de clase ).

### Componentes funcionales

- En las primeras lineas importa lo que usará el componente como librerias u otros componentes
- Luego define una función que ejecutará el codigo adecuado segun su tarea, pudiendo incluso mostrar otro componentes dentro
- La funcion del componente debe exportarse para que pueda ser utilizada en otras partes de la aplicación.

### Componentes de clase

- En las primeras líneas, se importan las librerías o componentes que el componente utilizará.
- Luego, se define una clase que extiende la clase base 'React.Component'. Dentro de esta clase, se pueden definir propiedades y métodos que gestionarán el comportamiento del componente.
- La clase de componente debe exportarse para que pueda ser utilizada en otras partes de la aplicación.

> la extension de VisualStudioCode "ES7+ React/Redux/React-Native/JS snippets proporciona *snippets* muy útiles, por ejemplo el snippet *rafce* genera una plantilla de componente de funcion flecha con export (React Arrow Function) y MUCHOS otros snippets que puedes consultar en la Documentación de ES7+.

> en el desarrollo de este proyecto usaremos componentes de tipo funcional.

### Construccion de un componente

```
import { NavLink } from "react-router-dom";
import style from "./NavBar.module.css";

const NavBar = () => {
    return (
        <div className={style.NavBar}>
            <NavLink to="/home"> Home </NavLink>
            <NavLink to="/detail"> Detail </NavLink>
            <NavLink to="/create"> Create </NavLink>
        </div>
    );
};
export default NavBar;
```

```css
.Navbar {
    display: flex;
    flex-direction: row;
    margin: 0;
    padding: 0.5rem;
    width: 100%;
    height: 2rem;
    justify-content: space-around;
    z-index: 9;
    border-radius: 0.5rem;
    border: 1px solid black;
}
```

tenemos un componente con funcionalidad y estilos, que NO recibe ni envía ningun *dato*, y que puede ser importado dentro de otro componente, o directamente por App.jsx para mostrarlo siempre por sobre cualquier componente.

diseñemos un componente que reciba datos por props.

> las props, son datos que el componente padre le puede dar a un componente hijo en forma de atributos html

```jsx
import { NavLink } from "react-router-dom";
import style from "./Card.module.css";

const UserCard = (props) => {
    return (
        <div className={style.UserCard}>
            <NavLink to="/detail"> {props.name} </NavLink>
            <p>eMail:{props.email}</p>
        </div>
    );
};
export default UserCard;
```

```css
.UserCard {
    display: flex;
    flex-direction: column;
    margin: 0.5rem;
    padding: 0.5rem;
    height: 2.5rem;
    justify-content: space-evenly;
```

```
    border-radius: 0.5rem;
    border: 1px solid black;
}
```

tenemos un componente con funcionalidad y estilos, que SI recibe *datos* por props y NO envía ningun *dato*

diseñemos un componente que envíe datos por props.

```
import { NavLink } from "react-router-dom";
import style from "./Card.module.css";

const CardsContainer = () => {
const users = [{},{},{}]; /*esto es un ejemplo de un array con muchos
usuarios*/

// lo ideal es que el array de users se obtenga desde un estado global
//const users = useSelector(state=>state.users);
//para usar useSelector debemos importarlo desestructurado desde react-redux

    return (
        <div className={style.CardsContainer}>
            {users.map(user=>{
                return <UserCard
                id={user.id}
                name={user.name}
                email={user.email}
                phone={user,phone}
                >
            })}
        </div>
    );
};
export default CardsContainer;
```

```
.CardsContainer {
    display: flex;
    flex-direction: row;
    width: 100%;
    margin: 0.5rem;
    padding: 0.5rem;
    justify-content: space-evenly;
    border-radius: 1rem;
    border: 1px solid black;
    flex-wrap: nowrap;
}
```

tenemos un componente con funcionalidad y estilos, que SI recibe *datos* pero aún no hemos determinado como va a recibir el array de usuarios, y por cada usuario, genera un componente UserCard al cual le envia los datos del usuario correspondiente.

Para determinar el origen del cual vendrá el array de usuarios, implementaremos la incorporacion del estado global a manos de Redux (ver implementación de redux).

## Ciclo de vida de un componente

**en componentes de clase**

se trabajan los ciclos de vida con component did mount, component did update y component dismount.

**en componentes funcionales**

se Emulan las funciones de cdm, cdu y cd usando distintas partes del hook useEffect

dentro de nuestro componente, importaremos use effect desde react:

```
import { useEffect } from "react";
import { CardsContainer } from "./components";

const Home = () => {
    //en este primer useEffect podemos ver la aplicacion de funciones que se
disparan cuando el componente se monta y cuando se desmonta
    useEffect(() => {
        console.log("se ha montado el componente");
        return () => {
            console.log("se ha desmontado el componente");
        };
    }, []);

    //en este segundo useEffect podemos ver la aplicacion de funciones que se
disparan cuando el componente se actualizan, es decir, cuando alguna de las
propiedades de su array de dependencias sufre cualquier cambio.

    useEffect(() => {
        console.log(
            "se ha actualizado el componente por la modificacion de una
dependencia"
        );
    }, [dependecia1, dependencia2]);

    return (
        <div className={style.Home}>
            <CardsContainer />
        </div>
    );
```

```
};
export default Home;
```

## Implementación del estado global con Redux

1. dentro de src, debemos crear una carpeta que podemos llamar 'redux'. y dentro de ella:
2. creamos el archivo store.js.
3. creamos el archivo reducer.js
4. creamos el archivo actions.js
5. creamos el archivo actionTypes.js
6. darle el provider de react-redux a nuestro archivo main.jsx de la carpeta src.
7. determinar que componentes despacharán actions para modificar el estado global y determinar en que etapa del cilco de vida lo haran. (implementacion de useEffect y useDispatch)
8. determinar que componentes consumirán el estado global

### store.js

esta definicion del store de redux, se puede usar *tal y como está en el ejemplo* para distintos proyectos.

```
import { createStore, applyMiddleware, compose } from "redux";
import rootReducer from "./reducer";
import thunkMiddleware from "redux-thunk";

const composeEnhancer = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ ||
compose;

const store = createStore(
    rootReducer,
    composeEnhancer(applyMiddleware(thunkMiddleware))
);
export default store;
```

### reducer.js

esta definicion del store de redux, se puede usar *tal y como está en el ejemplo* para distintos proyectos.

```
import { GET_USERS,GET_USER,FILTER_USERS_BY_SOURCE } from "./actionTypes";
const initialState = {
    users=[],
    user=[],
};

const rootReducer=(state=initialState, action)=>{
    switch(action.type){
```

```
        case GET_USERS:
        return { ...state, users:action.payload };
        case GET_USER:
        return { ...state, user:action.payload };
        case FILTER_USERS_BY_SOURCE:
        return { ...state, users:action.payload };
        default: return{ ...state};
    }
};
export default rootReducer;
```

### actionTypes.js

los action types son simplemente una forma de lograr que nuestro editor de texto nos avise si hemos escrito mal el nombre de una action, esto se logra gracias a que guardamos el string que representa el nombre de la action, dentro de una variable con un nombre igual, de modo que al ser una variable, el editor de codigo nos avisa si esta mal escria y tambien nos da recomendaciones de autocompletado.
las action deben exportarse para ser importadas y usadas en las actions.

```
export const GET_USERS = "GET_USERS";
export const GET_USER = "GET_USER";
export const FILTER_USERS_BY_SOURCE = "FILTER_USERS_BY_SOURCE";
```

### actions.js

Las actions son funciones que devuelven una funcion asincrónica, la cual se encarga de darle la orden al reducer de manipular el estado.
la instruccion que recibe el reducer es el dispatch, que contiene el type y opcionalmente el payload.

para ser mas específicos, la funcion externa se llama action creator, y le interna es en si, la action.

todas deben exportarse para ser usadas por el reducer

```
import { GET_USERS, GET_USER, FILTER_USERS_BY_SOURCE } from "./actiontypes";

export const getUsers=(){
    return async function(dispatch){
    const apiData = await axios.get("endpoint al quiero pedir/users");
    const users = apiData.data.length ? apiData.data : [];
    dispatch ({ type: GET_USERS, payload: users });
    };
};

export const getUser=(id){
```

```
    return async function(dispatch){
    const apiData = await axios.get(`endpoint al quiero pedir/users/${id}`);
    const user = apiData.data.length ? apiData.data : {};
    dispatch ({ type: GET_USER, payload: user });
    };
};

export const filterUserBySrc=(src){
    dispatch ({ type: FILTER_USERS_BY_SOURCE });
};
```

### provider

para proveerle a toda nuestra app de un estado global,en main.jsx
anidamos todo dentro de la etiqueta provider de ract-redux,
y al provider le damos como *prop* nuestro store, importado de la carpeta store.

```
import React from "react";
import ReactDom from "react-dom/client";
import App from "./App";
import { BrowserRouter } from "react-router-dom";
import { Provider } from "react-redux";
import store from "./redux/store";

ReactDOM.createRoot(document.getElementById("root")).render(
    <Provider store={store}>
        <BrowserRouter>
            <App />
        </BrowserRouter>
    </Provider>
);
```

### flujo de redux

```
import { useEffect } from "react";
import { CardsContainer } from "./components";
import { useDispatch } from "react-redux":
import { getUsers } from "../../redux/actions"

const Home = () => {
    const dispatch = useDispatch();
    useEffect(() => {
        dispatch(getUsers());
    }, [dispatch]);
```

```
// cuando usamos dispatch como una variable que guarda la ejecucion de
useDispatch, y usamos esa variable dentro de un useEffect, conviene siempre
agregar la variable dispatch al array de dependencias para librarnos de
errores en la consola.
    return (
        <div className={style.Home}>
            <CardsContainer />
        </div>
    );
};
export default Home;
```

en este ejemplo, podemos ver como después de haber hecho las configuraciones de redux, comienza el flujo de redux

1. **Dispatch**
   *el componente dispara el useEffect*
   Home se monta, disparando el useEffect. (tambien se puede disparar por ejemplo si sucede algo que modifique una de las dependencias del useEffect o si el componente se desmonta)
2. **Action**
   el action creator, que envía una funcion a modo de instrucción para el reducer.
3. **thunk**
   el middleware intercepta esta instruccion antes de ser recibida por el reducer y , ejecuta la action.
4. el **reducer**
   recibe la instruccion y determina cual es el action type que debe ser ejecutado, *reemplazando* el estado por uno nuevo, que generalmente es una copia del antiguo estado con alguna modificacion según la action y el payload lo determinen.

en este ejemplo, el componente home, realiza cambios en el estado, pero no lo consume, esperando que sea su componente hijo y cualquier otro componente suscrito al estado global, quienes consuman el contenido del estado global despues de el montaje de home.