yarn add @material-ui/core

<link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap" />

Plug in

Eslint PARA VALIDAR CODIGO INCORRECTO. NO PERMITE UN CHEQUEO ESTATICO DEL CODIGO.

**KEY DE LOS ARRAY PARA EL RENDERIZADO**

Cualquier componente que se genere dentro de un array necesita una key para el renderizado sea más eficiente. Se puede usar como key el índice del array. Pero si queremos agregar un componente o reorganizar uno no se va a resolver el problema del renderizado (va a ser lo mismo). Por eso es mejor que sea una clave NATURAL, algo que sabemos que no se repita (no depende de la ubicación del componente dentro de array como el índex

**COMUNICACIÓN ENTRE COMPONENTES**

Si queremos la comunicación entre componente (ósea que cuando aplicamos una acción sobre uno este se lo comunique a otro) podemos hacerlo de diferentes formas:

* Burbujeo: un componente a través de un evento le manda a este evento a tu componente padre y a partir de este componente padre se lo manda al correspondiente… o depende donde se ubique el componente va a ser el recorrido que va a realizar. Muy complejo, es engorroso.
* url: al seleccionar un componente se altera la URL, entonces hacemos que el componente sea reactivo frente al cambio de URL. No hay una comunicación directa ente componentes.
* Redux: podemos establecer un estado de la aplicación y así se mostrarían los valores. Después lo vamos a ver bien.

**PARA METER ESTILOS Y RESPONSIVE**

* React-bootstrap: https://react-bootstrap.github.io/
* React-Flexbox-Grid: <https://roylee0704.github.io/react-flexbox-grid/>

yarn add react-flexbox-grid

* Media Query

**CICLO DE VIDA DE RACT**

1. Se ejecuta el constructor de nuestro componente principal (app.js). Estableciendo el STATE con =…. “state =”
2. Render () {} con código JSX.
   1. Componentes nuestros
      1. Constructor que puede ser vacío o no (opcional)
      2. Render (si o si)
   2. Componentes h1, div, a
      1. Constructor que puede ser vacío o no (opcional)
      2. Render (si o si)
3. ---------------------------------SE CARGA LA PANTALLA------------------------------------
4. Puede surgir un evento que provoque una nueva renderizacion del componente. Una de estas es

This.setState => nuevo render

1. Provoca el renderizado de app.js y luego todos los demás.

*“También si se cambia una propiedad se fuerza el renderizado del componente”*

*shoulComponentUpdate() 🡪 función que permite una mejora de la eficiencia*

**FASES:**

* Montaje
* constructor
* getDeriverGetPromp
* render (genera los elementos mostrados en pantalla)
* componentDidMount (es invocado luego que el componente es insertado en el árbol del componente del DOM. Se usa para hacer peticiones.
* Actualización
* Render
* componentDidUpdate: invocado inmediatamente después que ocurre una actualización permite comparar valores de las propiedades antes o después de la actualización
* getDerivedStateFromProps
* ShouldComponentUpdate
* getSnapshotBeforeUpdate
* Desmontaje: retirar del DOM
* componentWillUnmount

[*http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/*](http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)

*ACTUALIZACION*

* Cuando se aplica un setState
* Cuando se cambian las prop de un componente padre
* UpdateForce

**LIBRERÍAS**

* Moments (para la transformación de fechas) 🡪 <https://momentjs.com/>
* Material-UI: <https://material.io/> <https://material-ui.com/>
* FlexboxGrid
* ReactWeatherIcons

REDUX

Npm install –save redux

Plug in google REDUX DEVTOOLS

Primero lo que hacemos es crear una acción. La cual se la denomina actionCreator. Esta la creamos en otra carpeta dentro de SRC para independizar a todo componente. Esta acción de se debe pasar un objeto, el cual tiene un atributo “type” que sirve para identificarlo. Al servir para identificarlo es muy importante y por eso lo hacemos con la creación de una constante.

export const SET\_CITY = 'SET\_CITY';

export const setCity = value => ({ type: SET\_CITY, value });

Luego creamos un Store en otra carpeta dentro de src para mejorar la independencia de los componentes. Para crear el store tenes que importar createStore. El primer parámetro debe ser una función y el segundo es para que se vincule con google.

Después para disparar esa acción se usa Store.dispach(“actionCreator”).

import { createStore } from 'redux';

export const store = createStore(() => {}, window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_());

A este Store lo vamos a importar desde el archivo index.js el cual es para que este el store disponible en todos los componentes.

import { store } from './store';

ReactDOM.render(

    <Provider store={store}>

        <App />

    </Provider>, document.getElementById('root'));

*¿Destructuring para permitir que todos los componentes puedan usar el STORE?*

Se define una función la cual retorna objetos que va a tener las funciones que va a invocar para hacer la creación de las acciones. Como definimos las funciones, acciones que se van a disparar. Con esta función que definimos, cuando quisiéramos llamar a una acción, la vamos a llamar con “this.props”.

Luego no vamos a retornar más el componente así solo, sino que vamos a exportar el resultado de la conexión (el cual es un componente con la habilidad de conectarse con el entorno. El segundo parámetro son las acciones que va a poder realizar el componente.

const mapDispatchToPropsAction =  dispatch => ({

  setCity: value => dispatch(setCity(value))

});

const AppConnected = connect(null, mapDispatchToPropsAction)(App);

export default AppConnected;

**SMART COMPONENTS VS DUMB COMPONENTS**

* Smart components (Containers): tienen acceso a toda la aplicación. Componente que tiene acceso al estado de la aplicación (esto se realiza con el connect). Para esto un componente va a ser un container cuando no tenga lógica de presentación. RECOMENDACIÓN: conteiners de alto nivel.
* Dumb Components (Presentationals): simplemente tienen el objetivo de hacer el renderizado sobre la pantalla. Tiene solamente lógica de presentación

**CAMBIO DE ESTADO**

Para realizar el cambio de estado debemos usar la función createStore, la cual recibe tres parámetros:

* El reducer. Primer parámetro es el estado al cual se va a cambiar. Esta es una función que recibe dos parámetros, el state anterior y la acción (actionCreator) y es por eso que es necesario importarla. En esta función a través de un SWITCH vamos a ver el tipo de acción que se está hablando y en base a eso vamos a retornas las diferentes cosas. En este caso se retorna directamente el atributo value del objeto action.
* Como segundo parámetro el estado inicial, que es el estado en que entes de cambiarlo.
* Tercer parámetro es para que podamos usar el redux de google.

export const SET\_CITY = 'SET\_CITY';

export const setCity = value => ({ type: SET\_CITY, value });

import { SET\_CITY } from './../actions';

export const city = (state, action) => {

    switch (action.type) {

        case SET\_CITY:

            return {...state, city: action.value}

    }

    return state;

}

const initialState = {

    city: 'Buenos Aires, ar',

}; //estadoInicial

export const store = createStore(city, initialState,

window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_());

Reducer: deben ser funciones puras. Ósea dependen solo de los parámetros de entrada, no dependen de los estados intermedios, ni acceso a la base de datos. Es una función predecible. Por eso no debemos alterar los valores que vienen pasados por parámetros.

NO ALTERAR EL ESTADO. Cuando hacemos {…state, prop: ‘nuevo valor’} es como que hacemos una copia, no alteramos nada.