Curso de React Router y Redux

# ¿Qué es React Router y cómo instalarlo?

Si estás empezando la escuela de JavaScript desde este curso deberás crear una copia del repositorio de Platzi Video:

git **clone** **https**://github.com/platzi/PlatziVideo.git

Una vez tienes listo el repositorio vas a crear una nueva rama para trabajar en ella a lo largo del curso:

git checkout -**b** feature/router-redux

Ya que nos encontramos dentro de la rama vamos a instalar React Router, la librería que nos va a permitir manejar rutas dentro de nuestra aplicación:

npm **install** react-router-dom --save

# ¿Qué es React Router y cómo instalarlo?

Dentro de nuestro proyecto vamos a crear una carpeta llamada routes donde vamos a ir añadiendo las rutas que necesitemos en la aplicación.

Las rutas que añadamos debemos definirlas con el componente Route y estas deben estar encapsuladas dentro del componente BrowserRouter del paquete de react-router-dom. Para definir una ruta con el componente Route debemos pasarle las props de:

* path para indicar la url.
* exact si queremos que funcione única y exactamente con la url que le digamos.
* component para indicarle el componente que va a renderizar.

npm run start

json-server initialState.json

Recien tuve un tipo en App.js

**import** React **from** "react";

**import** { BrowserRouter, Route } **from** "react-router-dom";

**import** Home **from** "../containers/Home";

**const** App = () => (

<BrowserRouter>

<Route exact path="/" component={Home} />

</BrowserRouter>)

export default App;

**es con parentesis, no con llaves**. Tarde 15 minutos en darme cuenta jaja!

Yo tuve un pequeño problema con el json-server porque no lo tenía instalado globalmente, ejecutando el siguiente comando sirvio perfecto todo lo demás.

npm **install** -g **json**-**server**

Tomado de: <https://www.npmjs.com/package/json-server>

# Container: Login

Vamos a descargar el proyecto del curso de Frontend de escuela de JavaScript para crear los componentes que necesitaremos, en caso de que vengas del curso de React no debes descargar nada.

Debemos modificar nuestra configuración del entorno de desarrollo local para que pueda funcionar con el uso de rutas, debemos ir al archivo webpack.config.js y añadir este fragmento de código antes de plugins:

module.exports = {

{/\*...\*/}

devServer: {

historyApiFallback: **true**,

},

{/\*...\*/}

}

Aqui esta el codigo de section para quien lo necesite

<section className='login'>

<section className='login\_\_container'>

<h2>Inicia sesión</h2>

<form className='login\_\_container--form'>

<input className='input' type='text' placeholder='Correo' />

<input className='input' type='password' placeholder='Contraseña' />

<button className='button'>Iniciar sesión</button>

<div className='login\_\_container--remember-me'>

<label>

<input type='checkbox' id='cbox1' value='first\_checkbox' />

Recuérdame

</label>

<a href='/'>Olvidé mi contraseña</a>

</div>

</form>

<section className='login\_\_container--social-media'>

<div>

<img src='../assets/google-icon.png' /> Inicia sesión con Google

</div>

<div>

<img src='../assets/twitter-icon.png' /> Inicia sesión con Twitter

</div>

</section>

<p className='login\_\_container--register'>

No tienes ninguna cuenta <a href=''>Regístrate</a>

</p>

</section>

</section>

# Container: Register

Para asegurar que nuestras rutas solamente se rendericen con la que haga match con la url debemos encapsular las rutas dentro del componente.

Atributos para los Route Objectos:

* path: la ruta en la que se renderizará el componente en forma de cadena de texto
* exact: un booleano para definir si queremos que la ruta tiene o no que ser exacta para renderizar un componente. Eg: /index !== /index/all.
* strict: un booleano para definir si queremos que el último slash sea tomado en cuenta para renderizar un componente. Eg: /index !== /index/.
* sensitive: un booleano para definir si queremos distinguir entre minúsculas y mayúsculas, y tomar esto en cuenta para renderizar un componente. Eg: /index !== /Index
* component: recibe un componente a renderizar. Crea un nuevo elemento de React cada vez. Esto causa que el componente se monte y desmonte cada vez (no actualiza).
* render: recibe un método que retorna un componente. A diferencia de component no remonta el componente.

La diferencia entre poner el switch y no ponerlo es que cuando tienes el mismo path para todos solamente toma el primero y lo renderiza.

# Container: 404 Not Found

Es importante siempre tener una ruta que renderice un componente para las urls que no existan, debemos añadir esta ruta al final del Switch para que sea el caso por default.

Fragment nos permite no añadir elementos extra al DOM, podemos utilizar Fragment de 2 maneras:

1. Añadiendo el componente o .
2. O implemente encapsulando nuestros elementos dentro de <>.

# Componente: Layout



# Manejando enlaces y configuraciones

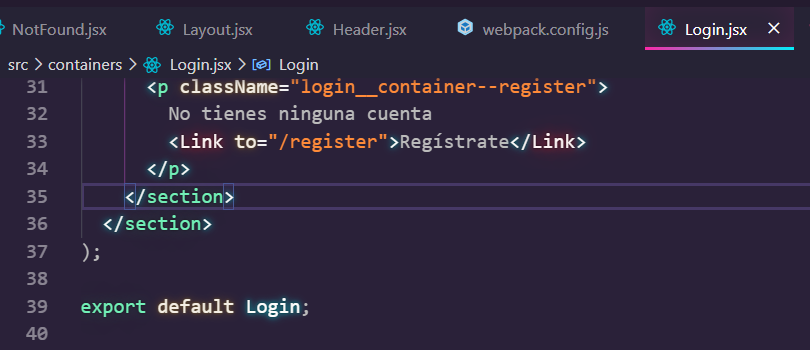
El componente es similar a un elemento ya que nos permite navegar dentro de la aplicación, pero sin la necesidad de tener que recargar la página. Para indicarle el destino a simplemente debemos pasarle la prop to='/mi-enlace'.

También es bueno porque no tiene limitaciones para darle parámetros, bueno para el SEO 😄

<Link to="/" rel="canonical">

<img src={Logo} widht="100" />

</Link>



¿Qué es Redux?

# ¿Qué es Redux?

Redux nos permite tener un contenedor predecible del estado en aplicaciones creadas con JavaScript, Nos ayuda a escribir aplicaciones que se comportan de una manera consistente, Esto significa que podemos utilizar esta lógica en aplicaciones del lado del cliente, trabajar del lado del servidor o crear aplicaciones para dispositivos móviles.

Uno de los principales uso que tiene Redux es con React pero puede ser implementado en cualquier librería o proyecto que este construido con JavaScript, lo cual incluye a Angular, Vue o algún otro framework o librería.

Redux nace de la arquitectura Flux, tomando inspiración del lenguaje funcional Elm y es creado por Dan Abramov y Andrew Clark en el 2015, Hoy en día es una de las librerías más utilizadas para el manejo del flujo de la información en aplicaciones.

Una de las principales motivaciones para crear Redux nace en resolver un problema y era el manejo del estado y el flujo de nuestras aplicaciones creadas en JavaScript. Redux propone una forma de manejar el estado donde podamos controlar cómo vamos a interactuar con otros elementos (llamadas a un API) o interacciones dentro de nuestra aplicación, teniendo en cuenta esto, Redux intenta de predecir las mutaciones que pueda sufrir el estado, creando restricciones de cuando y como pueden ser ejecutadas las actualizaciones en nuestras aplicaciones.

Redux es una librería muy pequeña que se puede incorporar en cualquier proyecto construido en JavaScript y se basa en tres principios:

### Única fuente de la verdad:

Nuestra aplicación solo debe de tener un único Store y es la única fuente de información.

### El estado es de solo lectura

La única forma de modificar el estado es emitiendo un acción, este objeto describe lo que va a ocurrir.

### Los cambios se realizan con funciones puras

Para realizar cambios al estado es necesario utilizar Reducers los cuales son funciones puras que toman el estado anterior, una acción y devuelve un nuevo estado con las modificaciones necesarias.

Teniendo en cuenta esta información continuaremos en el curso explicando cada uno de estos elementos que incorpora Redux en nuestra aplicación Platzi Video.

<https://carlosazaustre.es/como-funciona-flux>

# Instalación de Redux

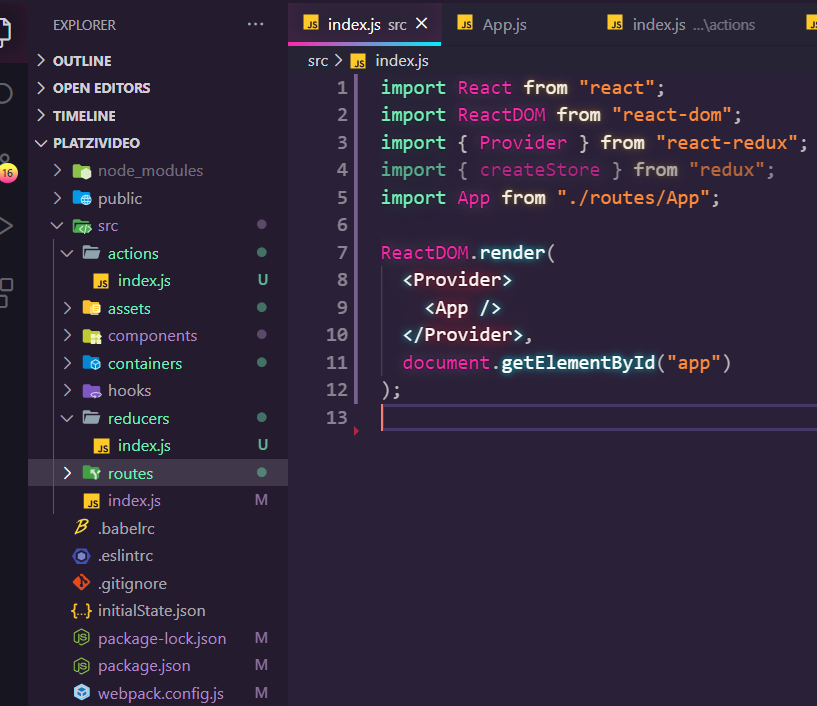
Vamos a instalar las dependencias para poder trabajar con Redux:

npm **install** redux react-redux --save

Dentro de nuestro proyecto vamos a crear una carpeta para nuestros **actions** y otra para los **reducers** que utilizaremos en Redux.

El paquete react-redux nos proporciona un Provider para poder encapsular nuestros componentes por medio de un connect para poder transmitir la información que necesitemos del

store a cada componente.



El flag -S es lo mismo que --save esto guarda para Producción  
El flag -D es lo mismo que --save-dev esto guarda para Desarrollo

## Provider:

**Recuerden que se tiene que encapsular nuestra aplicación dentro de un provider, porque nada fuera del provider podrá acceder al store**  
.  
El <Provider /> hace que la store de Redux esté disponible para cualquier componente anidado que se haya incluido en la función connect().  
.  
Dado que cualquier componente React en una aplicación React Redux se puede conectar, la mayoría de las aplicaciones mostrarán un <Provider> 'en el nivel superior, con el árbol de componentes completo de la aplicación dentro de él. . Normalmente, no puede usar un componente conectado a menos que esté anidado dentro de un<Provider>`.

# Creando el Store de Redux

Para crear un Store necesitamos llamar a la función createStore del paquete de redux pasándole los parámetros del reducer y initialState.

Para conectar un componente a Redux vamos a necesitar importar connect de react-redux, connect va a aceptar dos parámetros:

1. mapStateToProps: es una función que le va a indicar al provider qué información necesitamos del store.
2. mapDispatchToProps: es un objeto con las distintas funciones para ejecutar una action en Redux.

Esta clase se me complico mucho, por eso les dejo algunos principios de redux que me ayudaron entender mejor,

\*Esta basado en el patrón de diseño Flux

\*Toda la data de la aplicación, se encuentra en una estructura previamente definida.

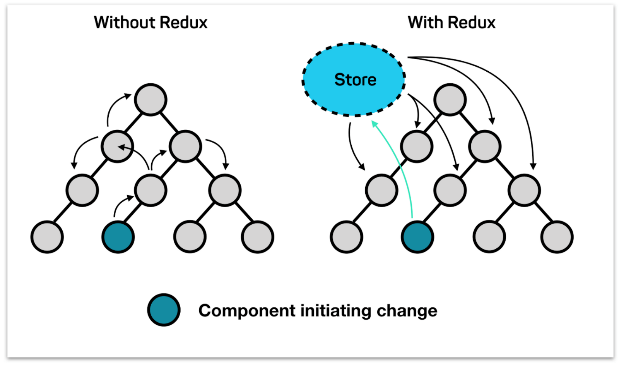
\*Toda la informción se encontrará almacenada en un único lugar llamado STORE.

\*El STORE jamas se modifica de forma directa.

\*Interacciones de usuario y/o código, dispara acciones que describen qué sucedió

\*El valor actual de la información de la aplicación se llama - State

\*Un nuevo estado es creado, en base a la combinanción del viejo estado y una acción, por una función llamada Reducer.

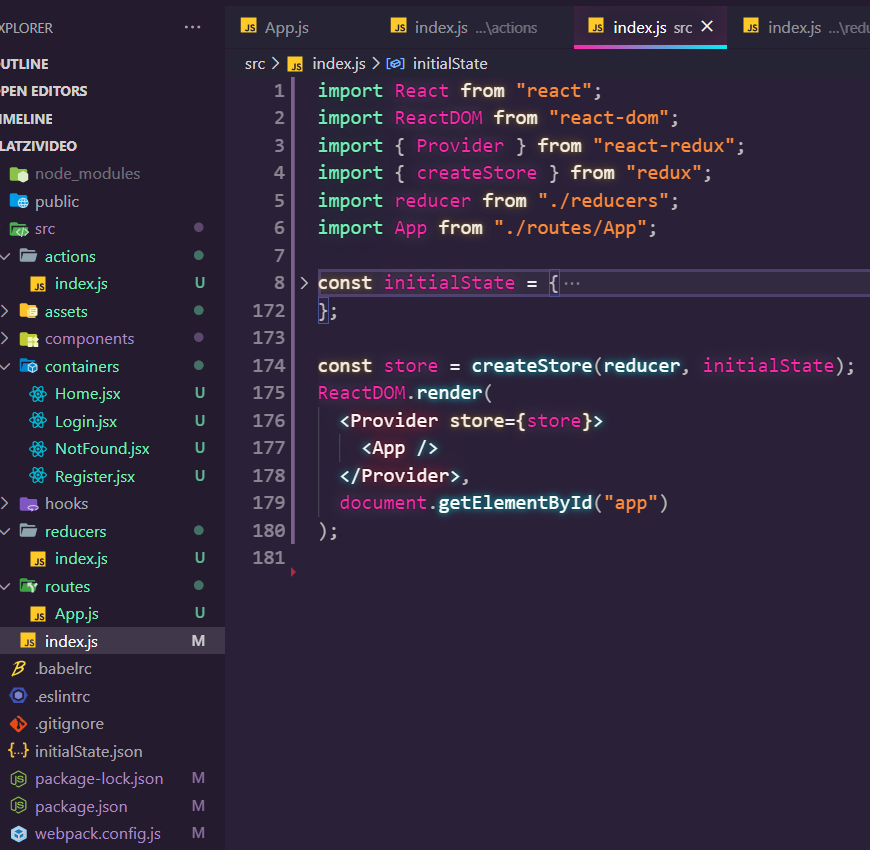


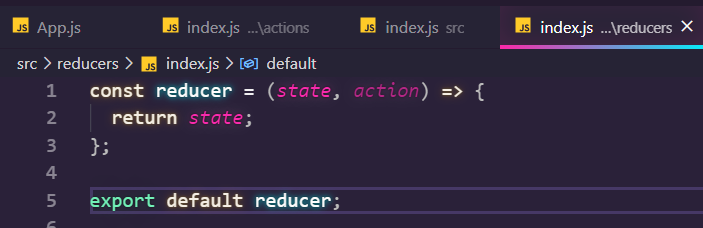
**Creando el Store de Redux**

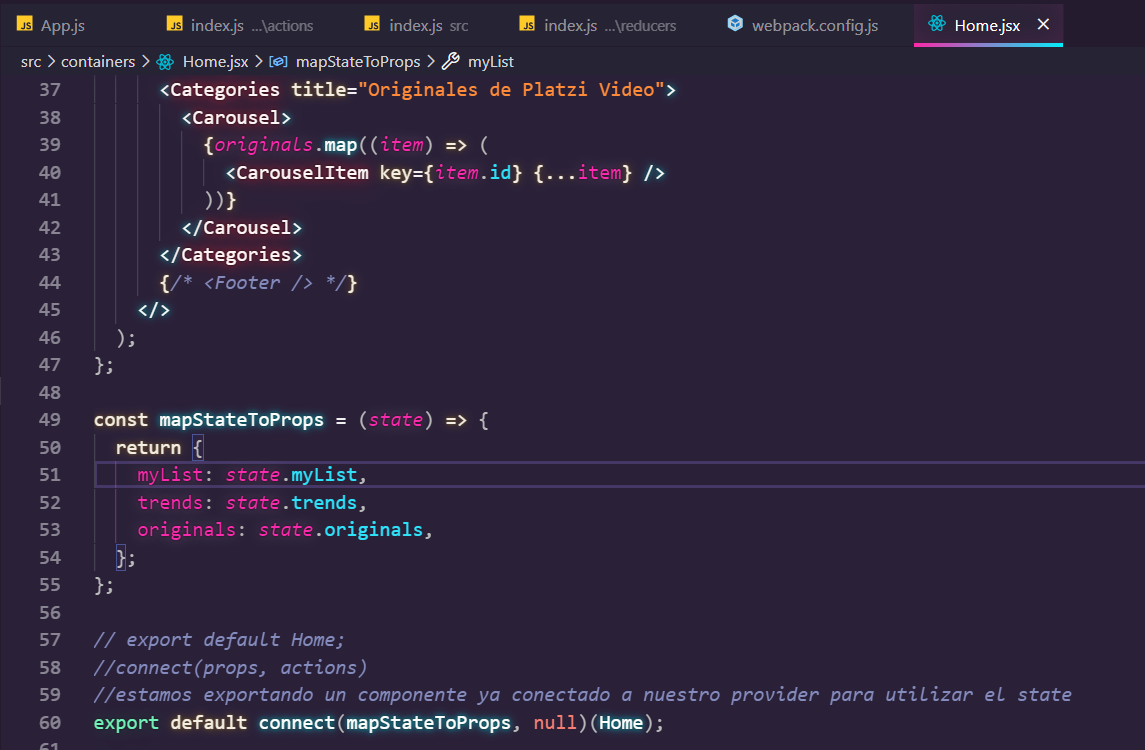
Para crear un Store necesitamos llamar a la función createStore del paquete de redux pasándole los parámetros del reducer y initialState.

Para conectar un componente a Redux vamos a necesitar importar connect de react-redux, connect va a aceptar dos parámetros:

mapStateToProps: es una función que le va a indicar al provider qué información necesitamos del store.  
mapDispatchToProps: es un objeto con las distintas funciones para ejecutar una action en Redux.





´

Aplicando Redux a nuestra aplicación

# Creando los reducers

Un action de Redux va a contener dos elementos:

* type: para indicar la acción que se va a ejecutar.
* payload: es la información que estamos mandando al reducer.

Dentro de los reducers usaremos un switch para separar la lógica por cada tipo de acción que tendremos en Redux.

Me tomó mucho tiempo entender Redux, tuve que volver a reproducir las clases y buscar en internet sobre arreglos y objectos porque no tenía ni idea de porque a veces destructurabas o por qué, y tampoco tenía idea de cómo funcionaban los datos y el estado pero ya por fin entendí.

<https://camo.githubusercontent.com/5aba89b6daab934631adffc1f301d17bb273268b/68747470733a2f2f73332e616d617a6f6e6177732e636f6d2f6d656469612d702e736c69642e65732f75706c6f6164732f3336343831322f696d616765732f323438343535322f415243482d5265647578322d7265616c2e676966>

<https://camo.githubusercontent.com/9de527b9432cc9244dc600875b46b43311918b59/68747470733a2f2f73332e616d617a6f6e6177732e636f6d2f6d656469612d702e736c69642e65732f75706c6f6164732f3336343831322f696d616765732f323438343739302f415243482d5265647578322d657874656e6465642d7265616c2d6465636c657261746976652e676966>

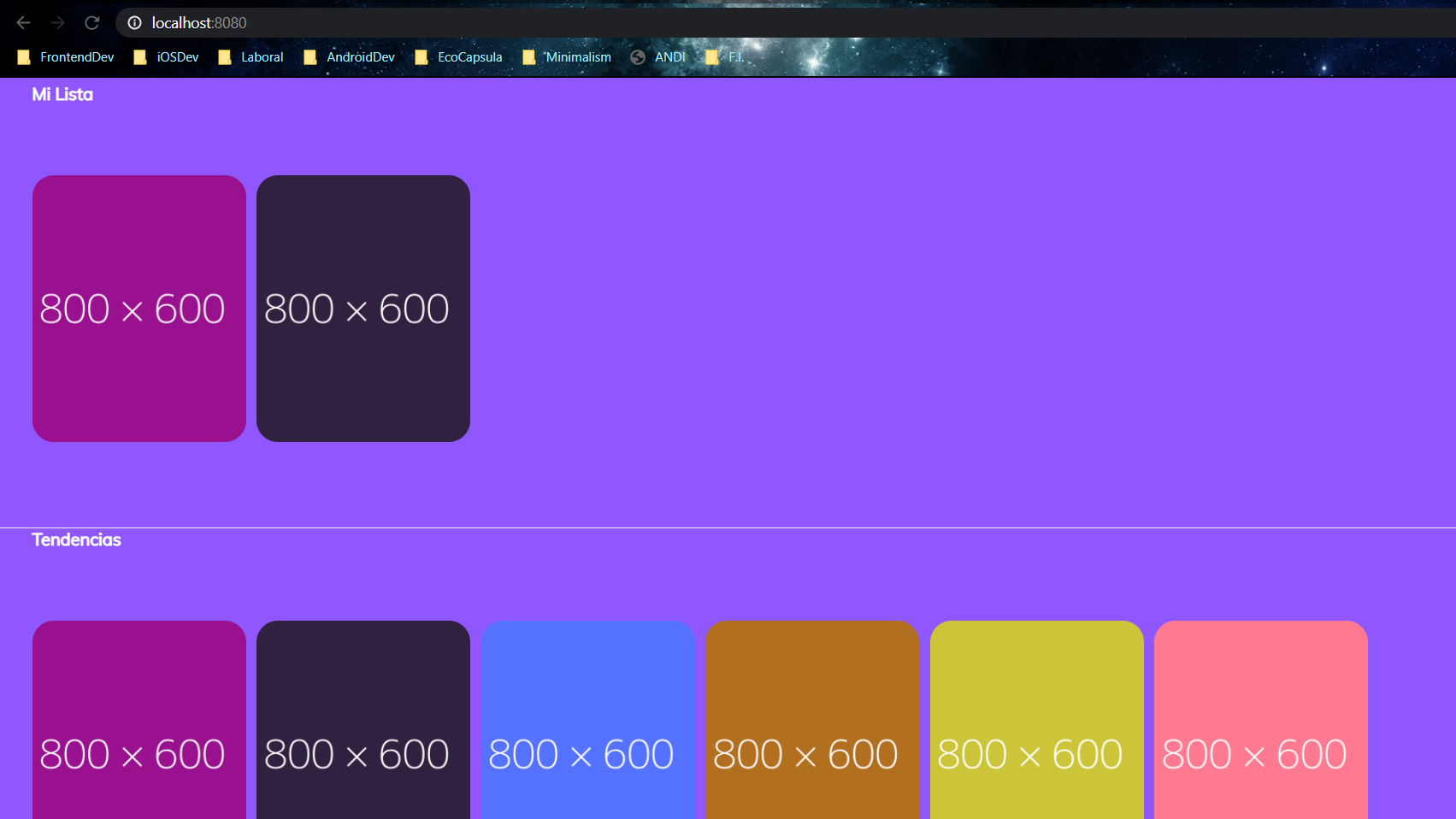
Después de repetir las clases, este es mi resumen:

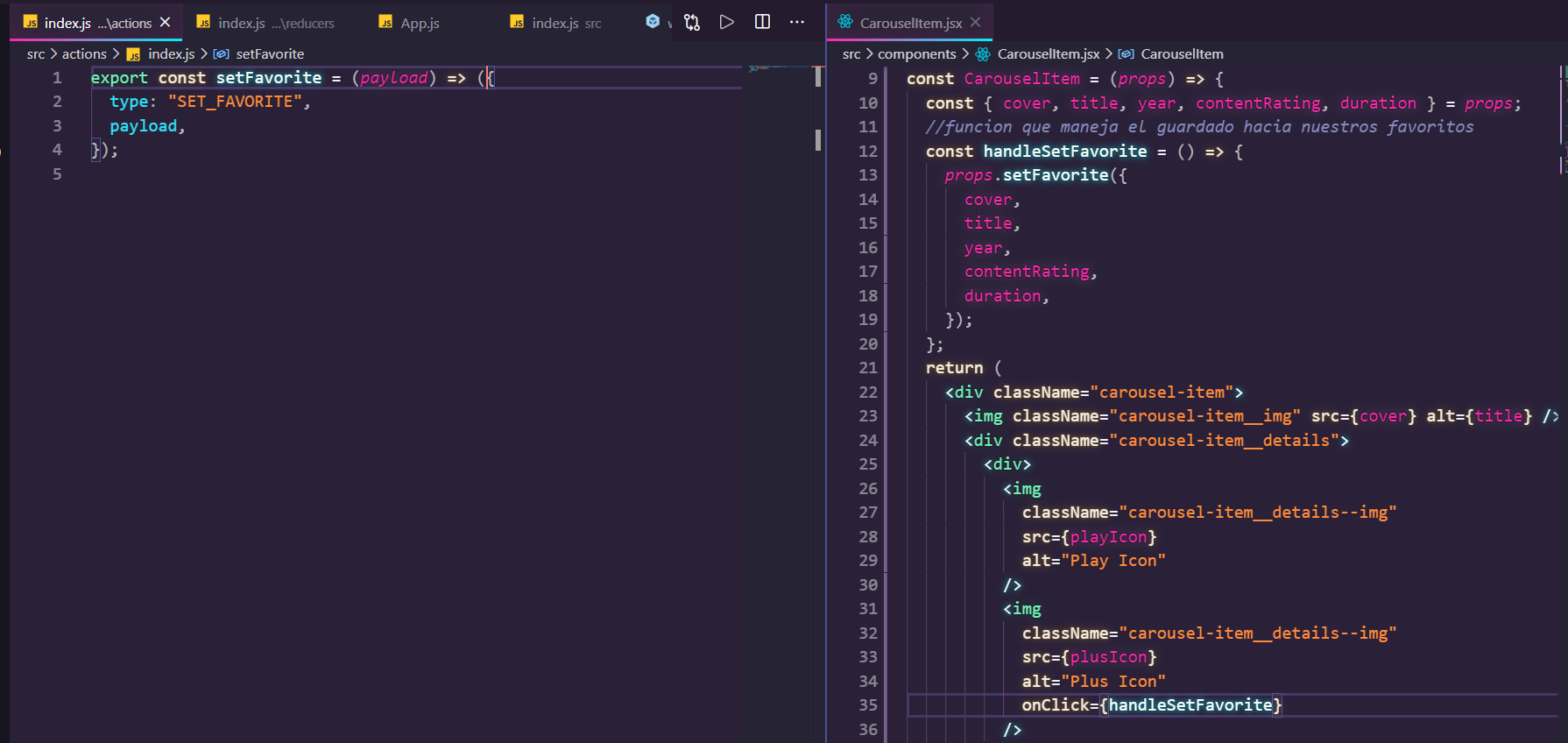
1. El Storage es como una base de datos de los estados
2. El connect permite la conexión entre el componente con el Storage y tiene 2 parámetros. Ejemplo - Connect (Primero,Segundo):

Primero: Envía el estado actual del Storage al componente para que sea utilizado por medio de su props. Para lograrlo, se crea una función que indicará que atributos se quieren utilizar en el componente y luego las asigna a su props.  
Segundo: Envía al Reduce la acción que hemos encapsulado y que, por alguna razón que aún no entiendo, al ser llamada por el import en el componente se le atribuye su props. Cuando esta acción llega al Reduce, es identificada mediante el Switch por su atributo Type para su ejecución. Luego, el Reduce envía el resultado al Storage para su actualización.  
**Importante**: El Reduce también tiene 2 atributos “(state, action)”:

* + state: Es el estado que el Stage envía al Reduce. Por lo tanto, es el estado que se va a actualizar.
  + action: Es la acción encapsulada que fue enviada por el componente al Reduce para que se actualice el estado del Stage.

Espero haberme dejado entender y que a alguien más se sirva para que siga aprendiendo sin muchas dudas en las siguientes clases.





Mencionar que si les surge un error de que cada item deberia tener una key, es porque no lo estamos mandando en el payload de setFavorite en CarouselItem, con esto lo pueden solucionar:

` const {id, cover, title, year, contentRating, duration } = props

const handleSetFavorite = () => {

props.setFavorite({

id,

cover,

title,

year,

contentRating,

duration

})

}

el problema sigue pasando si escoges dos veces el mismo item porque hay duplicidad de ids. En este caso ¿qué se puede hacer?

Corregié ese problema en modificar los key del archivo Home.jsx

<Search />

{mylist.length > 0 && (

<Categories title='Mi Lista'>

<Carousel>

{mylist.map((item) => (

<CarouselItem key={`${item.id}\_mylist`} {...item} />

))}

</Carousel>

</Categories>

)}

<Categories title='tendencias'>

<Carousel>

{trends.map((item) => (

<CarouselItem key={`${item.id}\_trends`} {...item} />

))}

</Carousel>

</Categories>

<Categories title='Originales de PlatziVideo'>

<Carousel>

{originals.map((item) => (

<CarouselItem key={`${item.id}\_originals`} {...item} />

))}

</Carousel>

</Categories>

Para elementos repetidos dentro del array, yo hice lo siguiente , modifiqué el reducer para filtrar si existe ese elemento dentro del state.myList.

case 'SET\_FAVORITE':

if (**state**.myList.filter((element) => element.id == action.payload.id).length !== 0) {

return **state**;

}

return {

...**state**,

myList: [...**state**.myList, action.payload],

};

12

Les comparto para evitar que se agregue un item duplicado al array mylist (Esta es mi manera de hacerlo tal vez haya alguna mejor por favor comentarla si es así.)

**const** { **id**, cover, title, year, contentRating, duration, setFavorite, mylist } = props

**const** handleSetFavorite = () =>{

**const** exist = mylist.find(item => item.id == **id**)

**if**(exist){

alert("Ya tienes agregado a favorito");

}**else**{

setFavorite({

**id**,

cover,

title,

year,

contentRating,

duration

})

}

}

recuerden agregar el mapStateToProps para acceder a mylist

const mapStateToProps = (**state**) => {

return {

mylist : **state**.mylist

}

}

export **default** connect(mapStateToProps, mapDispatchToProps)(CarouselItem)

# Finalizando reducers y eliminar favoritos

**Recomendación: Ulilizar constantes en Redux.**  
Es recomendable usar constantes en redux, por ejemplo, crear un archivo que se llame “actionTypes.js”, y dentro colocar lo siguiente:

**export** **const** SET\_FAVORITE = 'SET\_FAVORITE'

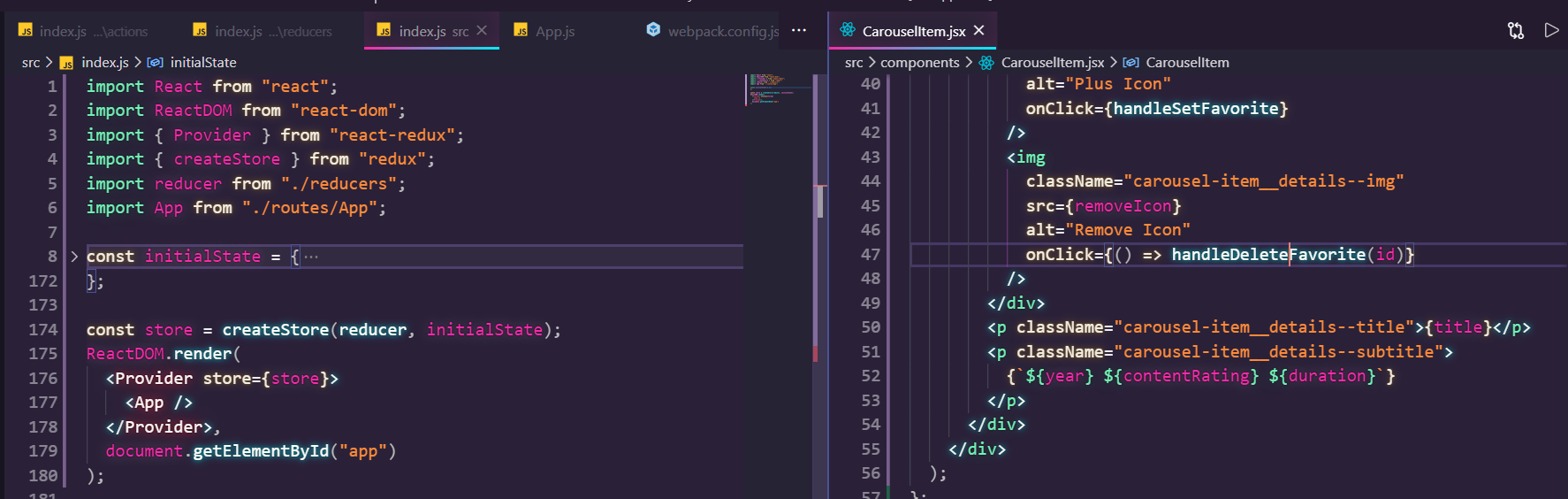
**export** **const** DELETE\_FAVORITE = 'DELETE\_FAVORITE'

Estas constantes las importaremos en los *actions* y en los *reducers*.

¿Con qué fin se usan estas constantes? Muchos no le encuentran sentido porque el valor tiene el mismo nombre que la constante, pero los beneficios son los siguientes.

* Ayuda a mantener una consistencia y evita los errores al escribir código, ya que si escribes *Strings* en las *actions* y en los *reducers* puede que en una de esas veces te equivoques.
* Algunas veces el usuario desea ver qué *actions* existen antes de trabajar en alguna. Si trabajas en un equipo puede que alguien la haya implementado. Tener todas las *actions* como constantes en un archivo te facilitará esa búsqueda.
* La lista de\_ action types\_ que fueron añadidas, eliminadas o actualizadas en un \_Pull Request \_ayuda a todo el equipo a visualizar rápidamente qué cambios se están haciendo en la funcionalidad.
* Si cometes un error (*typo*) cuando importas tus constantes, obtendrás undefined cuando intentes ejecutar tu código. Es más fácil encontrar este error de variable que intentando descifrar por qué no pasa nada cuando ejecutas tu *action*.

Espero que les haya servido de algo. No es algo que yo me haya inventado, muchos manejan esa escructura. Si gustan, aquí les dejo más información:



<https://platzi.com/clases/1613-redux/20655-archivos-typ-9/>

# Debuggeando nuestro proyecto (agregando validaciones a nuestro componente card)

Si quieres evitar que el mismo elemento se agregue muchas veces a la lista de favoritos:

reducers/index.js

case 'SET\_FAVORITE':

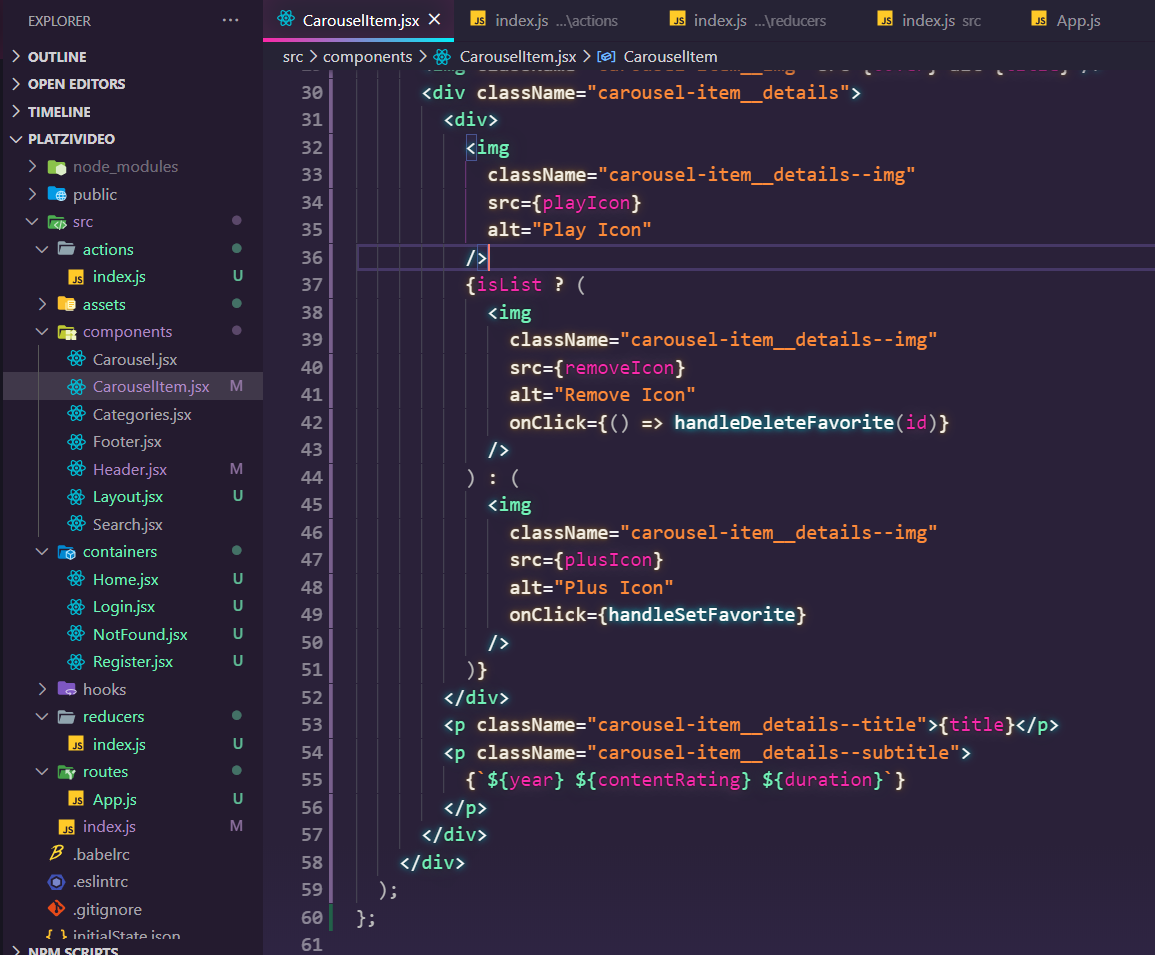
const exist = **state**.mylist.find(item => item.id === action.payload.id)

if (exist) return {...**state**}

return {

...**state**,

mylist: [...**state**.mylist, action.payload]

} 

**Hay otro bug** y es que una vez se agrega una película a la lista de mis **favoritos** ya no debe existir la opción de añadirla nuevamente.

Recordemos que **JS nos permite modificar los estilos de los elementos HTML**  
Para solucionar este bug agregue unas lineas en el método **handleSetFavorite** y \*\*handleDeleteFavorite \*\*, los cuales ahora removerán y agregaran el **display de none**, respectivamente.

const { id, cover, **title**, year, contentRating, duration, isList } = props;

const handleSetFavorite = () => {

props.setFavorite({ id, cover, **title**, year, contentRating, duration });

document.getElementById(id).style.display = 'none';

};

const handleDeleteFavorite = (itemId) => {

props.deleteFavorite(itemId);

document.getElementById(id).style.display = 'inline';

};

Adicionalmente, debemos agregar el atributo ID en cada imagen (plus and delete icon):

{isList ? (

<img

className='carousel-item\_\_details--img'

src={removeIcon}

alt='Remove Icon'

onClick={() => handleDeleteFavorite(id)}

/>

) :

(

<img

className='carousel-item\_\_details--img'

src={plusIcon}

alt='Plus Icon'

onClick={handleSetFavorite}

id={id}

/>

)}

# Crear Formulario de Login

Tengan en cuenta que su botón tiene que ser de tipo submit para que funcione.

Por si alguien se quedó con la duda:

**const** handleInput = **event** => {

setValues({

...form,

[**event**.target.name]: **event**.target.**value** // <---

});

}

Esta forma de inicializar un objeto dinámicamente se añade con ES6([Referencia](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer" \l "New_notations_in_ECMAScript_2015" \t "_blank))

Y sobre la función de actualización del estado:

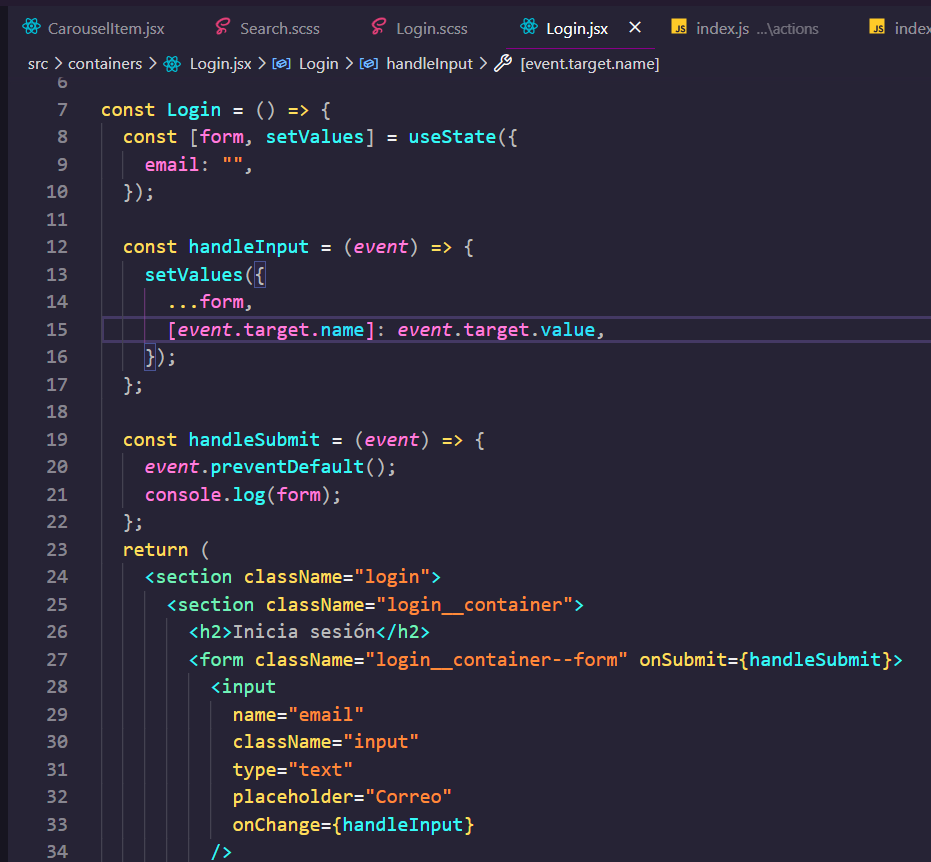
setValues({

...**form**,

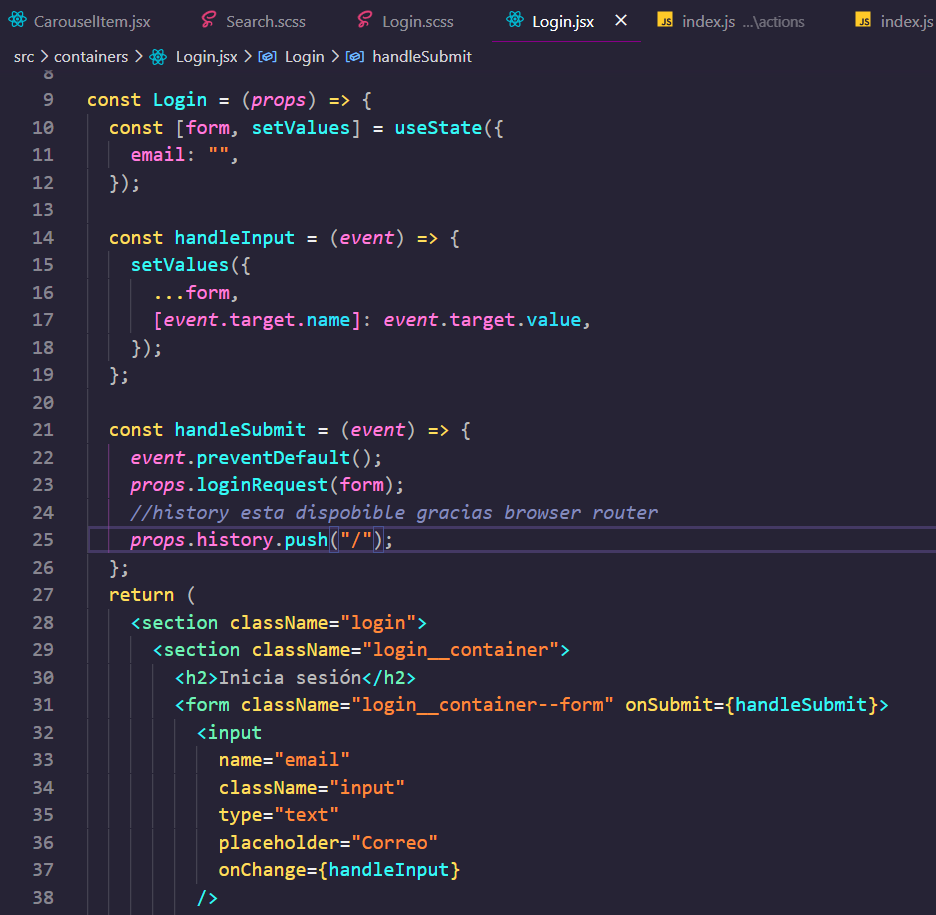
[event.**target**.**name**]: event.**target**.**value**

});

Al utilizar el operador de propagación, se está manteniendo el estado anterior y añadiendo uno nuevo, es decir, si tuviesemos ‘n’ inputs todos se almacenarían, para entender mejor este comportamiento, pueden comentar la línea del operador spread y ver como cuando se da submit solo se muestra el valor del último input modificado.



# Formulario de Login con Redux



# Creando un Servicio para Gravatar

Muchas veces la mejor opción no es descargar un paquete de npm ya que podemos ver la documentación, entender cómo funciona y nosotros implementar el código necesario para nuestro caso, ya que instalar todo el paquete puede volver más pesado nuestro proyecto.

Para nuestro servicio que llamará a Gravatar vamos a crear la carpeta utils y dentro añadir el archivo gravatar.js.

npm install md5 –save

