**01 HOLA REDUX**

Vamos a comenzar este laboratorio en el final del anterior, pero lo vamos a convertir en una aplicación React con Redux, para empezar a familiarizarnos con el concepto de Store.

Antes de nada y para que no se nos haga bola, vamos a introducir un refactor por legibilidad y mantenimiento del código. Pondremos cada subcomponente dentro de una carpeta individual

src/components/hello -- src/components/nameEdit

A continuación, crearemos un fichero index.ts bajo la carpeta components, donde iremos referenciando todos los componentes que vayamos creando y nos sea mas sencillo controlar las importaciones.

.src/components.index.ts

export { HolaReactComponent } from './hello/hello';

export { NameEditComponent } from './nameEdit/nameEdit';

Ahora actualizaremos las referencias en el app.tsx

import \* as React from 'react';

*// import { HolaReactComponent } from './components/hello';*

*// import { NameEditComponent } from './components/nameEdit';*

import {HolaReactComponent, NameEditComponent} from './components'

Ahora vamos a instalar los modulos de redux y react-redux junto con los typing que necesitan

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

**npm install @types/react-redux --save-dev**

Acto seguido vamos a crear nuestro modelo para el perfil de usuario, para ello crearemos la carpeta model bajo src y crearemos un archivo userProfile.ts

./src/model/userProfile.ts

export **interface** UserProfile {

firstname : string;

}

Ahora vamos a crear nuestro primer reducer, para almacenar el userProfile como un dato persistido en la Store. Primero creamos la carpeta reducers y a continuación el archivo userProfileReducer

./src/reducers/userProfileReducer.ts

import {UserProfile} from '../model/userProfile';

**const** defaultUserState : () **=>** UserProfile = () **=>** ({

firstname: 'Fernando Hierro'

});

export **const** userProfileReducer = (state = defaultUserState(), action) **=>** {

*// Despues tendremos un switch case para reemplazar el estado según necesidad*

return state;

}

Y ahora en nuestro index.ts, importaremos el reducer, le asignaremos el nombre con el que se va a guardar en el state, y lo tipamos en la interfaz que exponemos.

 ./src/reducers/index.ts

import { combineReducers} from 'redux';

import { userProfileReducer } from './userProfileReducer';

import {UserProfile} from '../model/userProfile'

export **interface** State {

userProfileState : UserProfile;

};

export **const** reducers = combineReducers<State>({

userProfileState: userProfileReducer

});

Ahora vamos a crear un Contenedor para el componente HelloReact para inicializarlo con el valor por defecto que hemos creado en el reducer. OJO !! Las propiedades y metodos que definimos en el Container para pasarle a nuestro componente a traves del container se tienen que llamar exáctamente igual y tienen que estar todas. De lo contrario no compilará

./src/components/hello/helloContainer.tsx

import \* as React from 'react';

import {connect} from 'react-redux';

import {HolaReactComponent} from './hello';

import {State} from '../../reducers/'

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

return {

username: state.userProfileState.firstname

}

}

**const** mapDispatchToProps = (dispatch) **=>** {

return {

}

}

export **const** HelloWorldContainer = connect(

mapStateToProps,

mapDispatchToProps

)(HolaReactComponent);

Vamos a deshabilitar el Componente NameEdit momentaneamente en nuestro fichero app.tsx, hasta el siguiente modulo donde introduciremos las actualizaciones en la Store y vamos a referenciar el container en lugar del Componente. Cambiaremos el contenido existente en el App.tsx por el código al pie, volviendolo de nuevo una constante .

import \* as React from 'react';

import {HelloWorldContainer} from './components/hello/helloContainer';

export **const** App = () **=>** {

return (

<>

<HelloWorldContainer />

</>

);

}

Ahora vamos a inicializar la store desde el punto de entrada de la aplicación, el main.tsx con el createStore de Redux y su propagación a traves del Provider

*// import \* as React from 'react';*

*// import \* as ReactDOM from 'react-dom';*

*// import { App } from '../src/app';*

*// ReactDOM.render(*

*// <App/>,*

*// document.getElementById('root')*

*// );*

import \* as React from 'react';

import \* as ReactDOM from 'react-dom';

import { createStore } from 'redux';

import { Provider } from 'react-redux';

import { reducers } from './reducers';

import { App } from '../src/app';

**const** store = createStore(reducers);

ReactDOM.render(

<Provider store={store}>

<>

<App />

</>

</Provider>,

document.getElementById('root'));

Vamos a comprobar que no hemos roto nada con tanta alteración !!

**02 – ACTIONS**

Vamos a recuperar la funcionalidad perdida de editar el Nombre, pero esta vez lo que haremos es hacer un update contra la Store.

Primero, vamos a crear una carpeta common donde crearemos un fichero para almacenar los diferentes tipos de acción que vamos a utilizar en nuestra aplicación.

./src/common/actionsTypes.ts

export **const** actionTypes = {

UPDATE\_USERPROFILE\_NAME : 'UPDATE\_USERPROFILE\_NAME '

}

En model, crearemos una clase para tipar nuestra acción, en el que le pasaremos dos propiedades, el tipo de acción que se coteja con el fichero de constantes y el payload, es decir los datos que pasamos a nuestro reducer. La intención de tiparlo es minimizar errores de sintaxis en la comunicación entre acciones y reducers.

./src/model/action.ts

export **interface** Action {

type : string,

payload:any;

}

A continuacion crearemos una carpeta actions para definir cada una de las interacciones que vayamos a hacer con el state y creamos nuestra nueva acción.

./src/actions/updateUserProfileName.ts

import {actionTypes} from '../common/actionTypes';

import { Action } from '../model/action';

export **const** updateUserProfileName = (newName : string): Action **=>** ({

type: actionTypes.UPDATE\_USERPROFILE\_NAME,

payload : newName,

});

Ahora vamos a manejar esta accion desde el reducer para actualizar la Store

./src/reducers/userProfileReducer.ts

import {UserProfile} from '../model/userProfile';

import {actionTypes} from '../common/actionTypes'

import { Action } from '../model/action';

**const** defaultUserState : () **=>** UserProfile = () **=>** ({

firstname: 'Fernando Hierro'

});

export **const** userProfileReducer = (state:UserProfile = defaultUserState(), action:Action) **=>** {

switch(action.type){

case actionTypes.UPDATE\_USERPROFILE\_NAME:

return handleUserProfileAction(state, action);

}

return state;

}

**const** handleUserProfileAction = (state: UserProfile, action:Action) **=>** {

return {

...state,

firstname: action.payload,

};

}

Ahora vamos a adaptar nuestro componente NameEdit al flujo de Redux como hicimos con el anterior. Creamos un container a su mismo nivel.

./src/components/nameEdit/nameEditContainer.tsx

import { connect } from 'react-redux';

import { NameEditComponent } from './nameEdit';

import {updateUserProfileName} from '../../actions/updateUserProfilename';

import { State } from '../../reducers'

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

return {

initialUserName: state.userProfileState.firstname

}

}

**const** mapDispatchToProps = (dispatch) **=>** {

return {

onNameUpdated: (name : string) **=>** dispatch(updateUserProfileName(name))

}

}

export **const** NameEditContainer = connect(

mapStateToProps,

mapDispatchToProps

)(NameEditComponent);

Vamos a actualizar las referencias del index de componentes para exportar los contenedores. También seria bueno hacerlo con el modelo. Aun asi las referencias absolutas a los archivos seguirán funcionando. Esto es una buena practica para exportar referencias de archivos similares

./src/components/index.ts

*// export { HolaReactComponent } from './hello/hello';*

*// export { NameEditComponent } from './nameEdit/nameEdit';*

export { HelloWorldContainer } from './hello/helloContainer';

export { NameEditContainer } from './nameEdit/nameEditContainer';

./src/model/index.ts

export { Action } from './action';

export { UserProfile } from './userProfile';

Modificamos nuestro app.tsx para que muestre ambos componentes.

import \* as React from 'react';

import {HelloWorldContainer, NameEditContainer} from './components';

export **const** App = () **=>** {

return (

<>

<HelloWorldContainer />

<NameEditContainer/>

</>

);

}

Y probamos.

Bonus track. En escenarios mas complejos estaría bien poder tener un control de las ejecuciones de las action y la actualizacion del state. Para eso tenemos algunas herramientas como el redux logger que implica instalación del modulo o Redux Dev Tool. Vamos a configurar este ultimo.

En Chrome instalar la siguiente extension:

<https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=es>

y en nuestro main.tsx introducimos el middleware

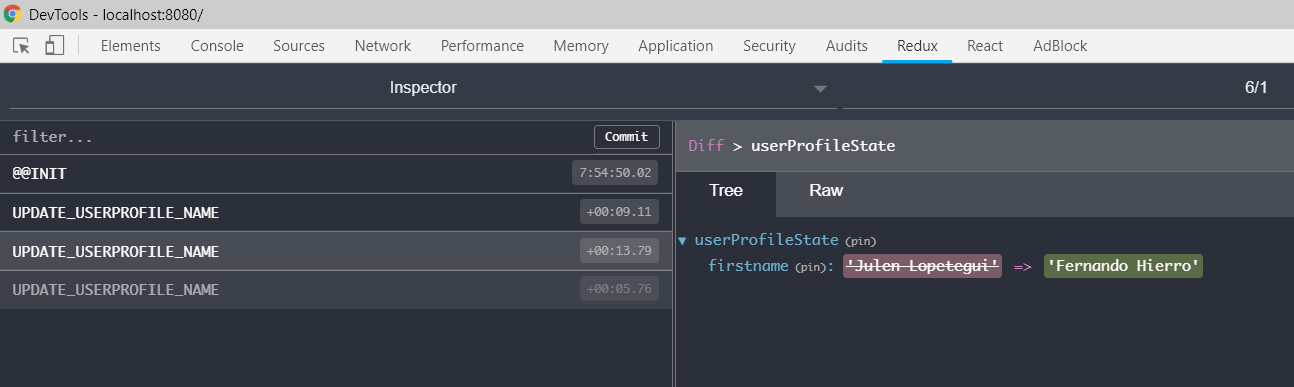
*//const store = createStore(reducers);*

**const** nonTypedWindow: any = window;

**const** middleware = nonTypedWindow.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && nonTypedWindow.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_();

**const** store = createStore(reducers,

middleware);



**02 – API**

Finalmente vamos a integrar las llamadas a endPoints externos para integrar sus datos dentro de nuestra aplicación. En primer lugar vamos a instalar redux-thunk que nos va a permitir tener asincronia en el contexto de nuestra aplicación

**npm install redux-thunk --save**

Despues de esto vamos a registrar Redux-thunk en main.tsx

./src/main.tsx

import \* as React from 'react';

import \* as ReactDOM from 'react-dom';

*//import { createStore } from 'redux';*

import {createStore, applyMiddleware, compose} from 'redux';

import reduxThunk from 'redux-thunk';

import { Provider } from 'react-redux';

import { reducers } from './reducers';

import { App } from '../src/app';

**const** nonTypedWindow: any = window;

*// const middleware = nonTypedWindow.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && nonTypedWindow.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_();*

*// const store = createStore(reducers,*

*// middleware);*

**const** composeEnhancers= nonTypedWindow.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

**const** store = createStore(reducers, composeEnhancers(

applyMiddleware(reduxThunk)

));

ReactDOM.render(

<Provider store={store}>

<>

<App />

</>

</Provider>,

document.getElementById('root'));

Vamos a pintar una lista de telefonos desde una api externa, creamos primero el modelo en esta ocasión lo vamos a crear como clase para generar un constructor para inicializar el estado.

export **class** Phone {

name: string;

brand: string;

color: string;

price: number;

imageUrl: string;

**constructor**() {

this.name = '';

this.brand = '';

this.color = '';

this.price = 0;

this.imageUrl = '';

}

}

Añadimos la referencia al Index del modelo

export { Action } from './action';

export { UserProfile } from './userProfile';

export { Phone } from './phone';

Ahora crearemos una clase api para el acceso a datos. Previa creación de carpeta.

./src/api/phones.ts

import { Phone } from '../model';

**const** GetAllPhones = (): Promise<Phone[]> **=>** {

**let** modeRequest: RequestMode;

**let** baseUrl = `http://reacttajamar.azurewebsites.net/`;

modeRequest = 'cors';

**let** obj = {

method: 'GET',

mode: modeRequest

};

**let** url = `${baseUrl}api/phones`;

return fetch(url, obj).then((response) **=>** parseJSON(response)).then(mapToPhoneList);

};

**const** parseJSON = (response: Response): any **=>** {

return response.json();

}

**const** mapToPhoneList = (phoneItem): Phone[] **=>** {

**let** result: Phone[] = [];

phoneItem.map((item) **=>** {

**let** phoneMap: Phone = {

name: item.name,

brand: item.brand,

color: item.color,

imageUrl: item.imageUrl,

price: item.price,

};

result.push(phoneMap);

});

return result;

};

export **const** phonesApi = {

GetAllPhones

};

Añadimos una nueva accion a nuestro fichero de constantes

export **const** actionTypes = {

UPDATE\_USERPROFILE\_NAME : 'UPDATE\_USERPROFILE\_NAME',

GET\_ALL\_PHONES: 'GET\_ALL\_PHONES'}

Añadimos una accion para traernos todos los telefonos, esta vez con una llamada a la api.

./src/actions/getPhoneList.ts

import { actionTypes } from '../common/actionTypes';

import { Phone, Action } from '../model/';

import { phonesApi } from '../api/phones';

export **const** phoneRequestAction = () **=>** (dispatch) **=>** {

phonesApi.GetAllPhones().then((result)**=>**{

dispatch(phoneRequestCompleted(result))

})

}

export **const** phoneRequestCompleted = (phones: Phone[]):Action **=>** {

return {

type: actionTypes.GET\_ALL\_PHONES,

payload: phones

}

}

Ahora vamos a crear el reducer y lo vamos a registrar en el index.

./src/reducers/phoneReducer.ts

import { Phone, Action } from '../model/';

import { actionTypes } from '../common/actionTypes';

**const** createEmptyPhoneList = (): Phone[] **=>** (

[]

);

export **const** phoneListReducer = (state = createEmptyPhoneList(), action: Action) **=>** {

switch (action.type) {

case actionTypes.GET\_ALL\_PHONES:

return handlePhonesCompleted(state, action);

}

return state;

};

**const** handlePhonesCompleted = (state: Phone[], action:Action) **=>** {

return action.payload

};

./src/reducers/index.ts

import { combineReducers} from 'redux';

import { userProfileReducer } from './userProfileReducer';

import {UserProfile, Phone} from '../model/'

import { phoneListReducer } from './phoneReducer';

export **interface** State {

userProfileState : UserProfile

phones: Phone[]

};

export **const** reducers = combineReducers<State>({

userProfileState: userProfileReducer,

phones:phoneListReducer

});

Igual que antes, vamos a crear un Container y despues el Componente que renderice la información que acabamos de recibir, y a su vez un subcomponente para hacer mas liviano el componente

import { PhoneListComponent } from './phoneComponent';

import \* as React from 'react';

import { connect } from 'react-redux';

import { State } from '../../reducers';

import { Phone } from '../../model/';

import { phoneRequestAction } from '../../actions/getPhoneList';

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

return {

phones: state.phones

}

};

**const** mapDispatchToProps = (dispatch: any) **=>** ({

loadPhones: () **=>** dispatch(phoneRequestAction())

});

export **const** PhoneListContainer = connect(

mapStateToProps,

mapDispatchToProps

)(PhoneListComponent);

Ahora creamos el componente con las Props que acabamos de definir en el Container

import \* as React from 'react';

import { Phone } from '../../model/';

export **interface** IPhoneListProps {

phones: Phone[];

loadPhones(): any;

}

export **class** PhoneListComponent **extends** React.Component<IPhoneListProps, {}> {

**constructor**(props) {

super(props);

this.handleClick = this.handleClick.bind(this);

}

**private** handleClick() {

this.props.loadPhones();

}

**public** render() {

return (

<div>

<input

type="submit"

className="btn btn-success"

value="CargarLista"

onClick={() **=>** this.props.loadPhones()}

/>

<table className="table table-striped">

<thead>

<tr className="bg-dark text-white">

<th>PHONE </th>

<th>MARCA</th>

<th>COLOR</th>

<th>PRECIO</th>

<th>IMAGEN</th>

</tr>

</thead>

<tbody>

{this.props.phones.map((item) **=>**

<tr>

<td>{item.name}</td>

<td>{item.brand}</td>

<td>{item.color}</td>

<td>{item.price}</td>

<td><img src={item.imageUrl} /> </td>

</tr>

)}

</tbody>

</table>

</div>

)

}

}

Primero, exponemos el container en el indice de componentes, despues vamos a cargar el Container en App.tsx para mostrarlo por pantalla

export { HelloWorldContainer } from './hello/helloContainer';

export { NameEditContainer } from './nameEdit/nameEditContainer';

export { PhoneListContainer } from './phones/phonesContainer';

import \* as React from 'react';

import { HelloWorldContainer, NameEditContainer, PhoneListContainer} from './components';

export **const** App = () **=>** {

return (

<>

<HelloWorldContainer />

<br/>

<NameEditContainer />

<br/>

<PhoneListContainer/>

</>

);

}

Probamos y vemos que las imágenes se nos están yendo mucho de madre, vamos a ver como cargar nuestros css.

Creamos la carpeta styles y añadimos nuestro archivo site.css

./src/styles.site.css

.fix-Width td>img {

max-width: 75px;

}

En el PhoneComponent, aplicamos la clase que acabamos de definir

{this.props.phones.map((item) **=>**

*// <tr>*

<tr className="fix-Width">

<td>{item.name}</td>

<td>{item.brand}</td>

<td>{item.color}</td>

<td>{item.price}</td>

<td><img src={item.imageUrl} /> </td>

</tr>

)}

Y en el main.tsx importamos la referencia

import { App } from '../src/app';

import './styles/site.css';

**const** nonTypedWindow: any = window;

Y probamos