REDUX (AND RTK) TUTO

CREDITS: TANIA RASCIA

Parte 1

Step by Step:

Creamos un nuevo proyecto:

npm init vite redux-tuto

Instalamos dependencias necesarias (*note que también está react-router-dom, aunque esta no pertenece al boilerplate de Redux en sí misma*).

npm i \

redux \

react-redux \

redux-thunk \

redux-devtools-extension \

react-router-dom

Entramos a la carpeta src y borramos todo:

cd src && rm \*

Luego creamos las carpetas donde alojaremos los archivos necesarios. Estos nombres son orientativos:

mkdir actions reducers components views

Y vamos a crear los archivos iniciales en la carpeta /src/

touch main.jsx main.css App.jsx

*Estos archivos pueden variar su nombre, en este caso, usamos* ***main*** *y la extensión* ***jsx*** *porque inicializamos con el bundler Vite…*

Así debería verse nuestra estructura hasta el momento:

└── src/

├── actions/

├── components/

├── views/

├── reducers/

├── App.jsx

├── main.css

└── main.jsx

Aquí tenemos algunas reglas para no perder tiempo con los estilos:

body {

margin: 0;

font-family: -apple-system, BlinkMacSystemFont, sans-serif;

-webkit-font-smoothing: antialiased;

-moz-osx-font-smoothing: grayscale;

overflow-y: overlay;

}

p {

font-size: 1.1rem;

}

nav {

display: flex;

padding: 2rem 0;

background: rgb(210, 209, 209);

position: sticky;

top: 0

}

nav section {

width: 100%;

}

nav a {

font-weight: 700;

padding: 1rem 1.5rem;

border-radius: 4px;

}

nav a:first-of-type {

margin-left: -1.5rem;

}

nav a:hover {

background: rgb(151, 148, 148);

color: white;

}

a {

color: #2e8417;

text-decoration: none;

margin: 1em 0;

}

section {

max-width: 800px;

margin-left: auto;

margin-right: auto;

padding: 0 1.5rem;

}

section h1 {

font-size: 3rem;

}

.post h2 {

font-size: 2.5rem;

}

.post-excerpt {

padding: 1.5rem 0;

border-bottom: 1px solid #eee;

}

.post-excerpt:first-of-type {

padding-top: 0;

}

.post-excerpt h2 {

margin-top: 0;

font-size: 1.5rem;

}

.comment {

background: #f0f0f0;

border-radius: 4px;

padding: 1rem 1.5rem;

margin: 1rem 0;

}

.button {

display: inline-block;

background: #044119;

color: rgb(245, 245, 245);

border-radius: 4px;

font-weight: 700;

padding: 0.75rem 1.5rem;

}

Antes de empezar a montar nuestro store de Redux, así se verán los archivos **App** y **main:**

App.jsx

import React from 'react'

const App = () => {

  return <h1>Redux (and RTK) tutorial</h1>

}

export default App

main.jsx

import React from "react";

import ReactDOM from "react-dom/client";

import App from "./App"

ReactDOM.createRoot(document.getElementById('root')).render(

  <React.StrictMode>

    <App />

  </React.StrictMode>

)

El siguiente paso en el camino es importar las dependencias que necesita Redux. Modificamos el archivo /src/main.jsx

// External

import React from 'react'

import ReactDOM from 'react-dom/client'

import { createStore, applyMiddleware } from 'redux'

import { Provider } from 'react-redux'

import thunk from 'redux-thunk'

import { composeWithDevTools } from 'redux-devtools-extension'

// Locals

import App from './App'

import rootReducer from './reducers'

// Styling

import './main.css'

const store = createStore(rootReducer, composeWithDevTools(applyMiddleware(thunk)))

ReactDOM.createRoot(document.getElementById('root')).render(

  <React.StrictMode>

    <Provider store={store}>

      <App />

    </Provider>

  </React.StrictMode>

)

Para que esto ‘despegue’ nos falta incorporar los reducers. Un **reducer** es una *función pura\*\** que determina cambios al estado que administra Redux. Es una función pura que retorna una copia del estado, incorporando los cambios que hayamos efectuado.

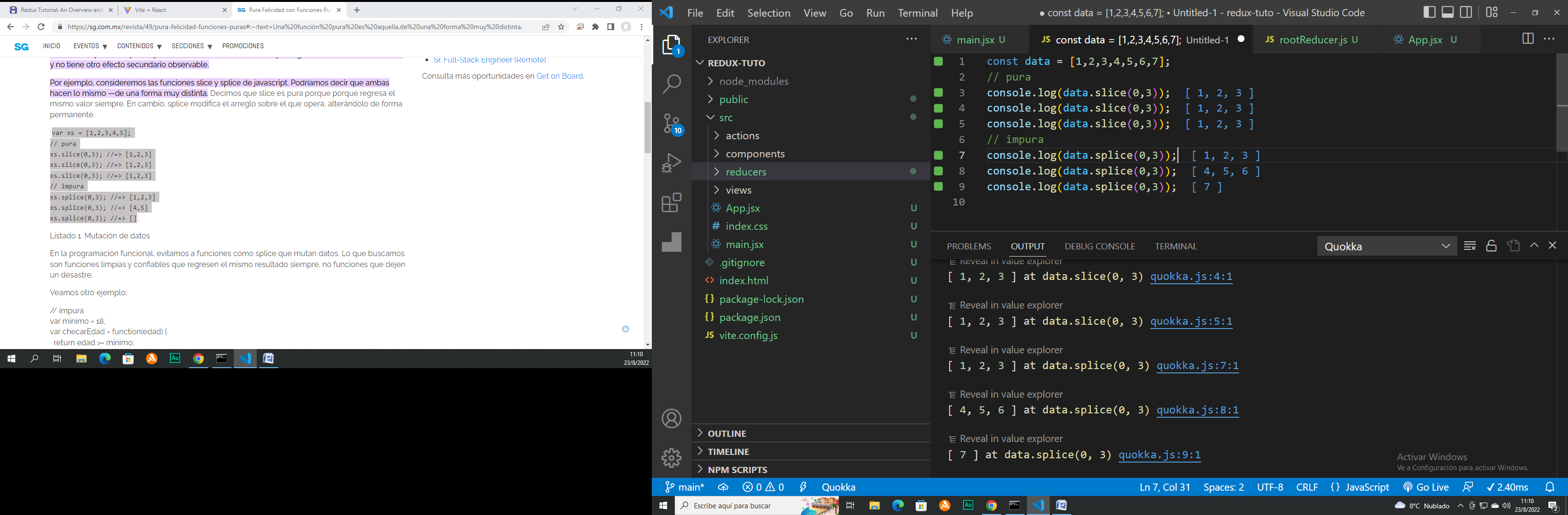
# Sabemos qué es una función pura?

Una función pura es aquella que, dada una misma entrada, siempre regresa el mismo valor de salida y no tiene otro efecto secundario observable. O sea, no modifica ningún objeto del “mundo exterior”.

Por ejemplo, los métodos slice y splice hacen algo parecido, pero de forma distinta.

**Slice** es una función **pura**: retorna el mismo valor siempre.

**Splice** modifica el arreglo sobre el que opera, alterándolo de forma permanente. Es un método **impuro**.



En la programación funcional preferimos evitar funciones impuras porque mutan los datos y el principio de inmutabilidad está fuertemente anclado al paradigma de la programación funcional. Esto es parte de algo más abarcativo, que conocemos como programación declarativa (vs. Imperativa).

Ok… retornemos a Redux y los reducers (funciones puras ok?)

Una cosa copada de Redux es que podemos tener muchos reducers y combinarlos en **root reducer** que va a usar el store, a través de **combineReducers**. Así podemos organizar el código en módulos pero siempre trabajaremos al final con todas esas referencias en un árbol de estado raíz.

Vamos a crear una especie de blog con una lista de posteos, y vamos a poner esos recursos en un **postsReducer**. El método **combineReducers** nos va a permitir agregar lo que vayamos necesitando (*Por ejemplo: comentarios ->* ***commentsReducer***).

Entonces, en la carpeta /src/reducers/ podemos crear el archivo que va a combinar todos nuestros reducers. En este caso le voy a poner main.js, pero eso queda a vuestro criterio. **main.js:**

import { combineReducers } from "redux";

import postsReducer from './postReducer'

const rootReducer = combineReducers({

  posts: postsReducer,

});

export default rootReducer;

Y, efectivamente, esto seguirá sin arrancar hasta que creemos el archivo **postsReducer** que estamos intentando importar… Irá en /src/reducers/

**postsReducer.js:**

export const initialState = {

  posts: [],

  loading: false,

  hasErrors: false,

}

export default function postsReducer(state = initialState, action) {

  switch (action.type) {

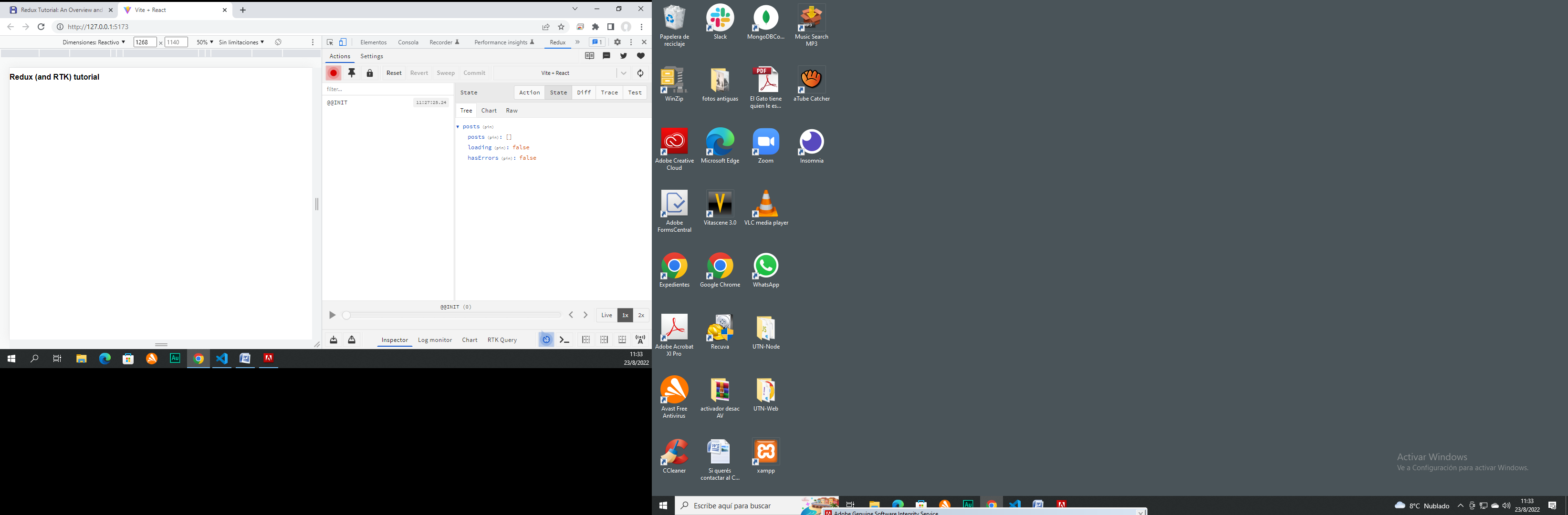
    default:

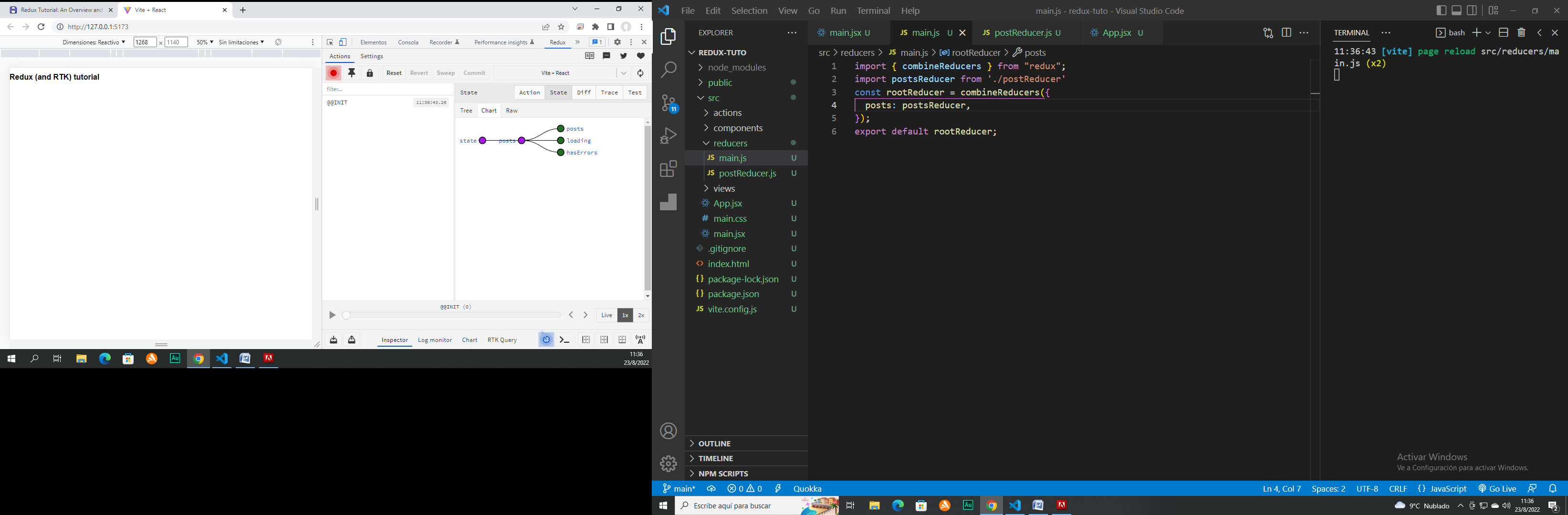
      return state

  }

}

Bueno… Terminar este scaffolding ha sido un tanto arduo, aunque nuestro proyecto ya llega a montarse. Ya podemos ver en las Redux DevTools que en el árbol de estado se encuentran las referencias que creamos en el **postReducer.**





Parte 2

Hasta aquí tenemos un **reducer** para los posteos, pero como no hemos creado ninguna acción (**actions**)… el reducer nos retorna siempre el estado sin posibilidad de modificarlo. Las **actions** nos proveen un modo de comunicarnos con el **store de Redux**. En la práctica, queremos obtener los recursos a través de una petición HTTP a una API y poner los datos en el **estado** que gestionamos con Redux.

Las peticiones HTTP son acciones especiales, por su naturaleza asincrónica. Por esta razón vamos a usar un Redux **thunk**. Si revisamos nuestros imports iniciales veremos que, previendo esto, ya lo importamos.

Crearemos el archivo src/actions/postsActions.js

Allí definiremos como constantes los tipos de acciones (**action types**). No es estrictamente necesario crearlas como constantes, pero es una práctica muy habitual por convención. Esto facilita la exportación de las acciones y previene *typos.*

Queremos tres tipos de acciones:

1. **getPosts**: indica a Redux que vamos a hacer la petición a la API
2. **getPostsSuccess**: Le pasa a Redux los datos si todo fue bien
3. **getPostsFailure**: Informa a Redux que hubo un error durante la llamada

Y así quedará nuestro **postsActions.js**

//Creamos los tipos de acciones (action types)

export const GET\_POSTS = 'GET\_POSTS'

export const GET\_POSTS\_SUCCESS = 'GET\_POSTS\_SUCCESS'

export const GET\_POSTS\_FAILURE = 'GET\_POSTS\_FAILURE'

/\* Luego creamos los 'action creators', funciones que retornan una acción, que consiste en un tipo y, opcionalmente, el payload que contiene información para actualizar el estado\*/

export const getPosts = () => ({

  type: GET\_POSTS,

});

export const getPostsSuccess = (posts) => ({

  type: GET\_POSTS\_SUCCESS,

  payload: posts,

});

export const getPostsFailure = () => ({

  type: GET\_POSTS\_FAILURE,

});

/\* Y creamos la el asynchronous thunk action que combina las tres acciones de arriba. Cuando invocamos este thunk, despacha la acción inicial getPosts() y le dice a Redux que se prepara para una API call. Entonces, en un bloque try/catch, hace lo necesario para obtener el recurso y despachar las acciones getPostsSuccess o getPostsFailure según se requiera.\*/

export function fetchPosts() {

  return async (dispatch) => {

    dispatch(getPosts())

    try {

      const res = await fetch('https://jsonplaceholder.typicode.com/postas')

      if (!res.ok) throw new Error()

      const data = await res.json()

      dispatch(getPostsSuccess(data))

    } catch (error) {

      dispatch(getPostsFailure())

    }

  }

}

Con esto hemos creado las acciones. A continuación, vamos a decirle a nuestro reducer que debe hacer con el estado en cada una de esas acciones.

Respondiendo a las acciones (modificamos postsReducer.js)

En **postsReducer** tenemos un **switch/case** que no hace nada por ahora… Así que vamos a importar las **actions** y las pondremos a trabajar en nuestro **reducer**:

import \* as actions from '../actions/postActions'

export const initialState = {

  posts: [],

  loading: false,

  hasErrors: false,

}

export default function postsReducer(state = initialState, action) {

  switch (action.type) {

    case actions.GET\_POSTS:

      return { ...state, loading: true };

    case actions.GET\_POSTS\_SUCCESS:

      return { posts: action.payload, loading: false, hasErrors: false };

    case actions.GET\_POSTS\_FAILURE:

      return { ...state, loading: false, hasErrors: true };

    default:

      return state;

  }

}

Con los reducers y las acciones terminadas y vinculadas, lo único que falta para ver la magia en movimiento es conectar todo a la aplicación.

Conexión de Redux con los componentes de React

Este proyecto usa algunas rutas, por eso necesitamos React Router. Recordemos que oportunamente lo instalamos con **npm** y lo importamos en el archivo **main.jsx**.

En esta etapa necesitamos crear un archivo **HomeView.jsx** y otro **PostsView.jsx** en la carpeta src/views

**HomeView.jsx:**

import React from 'react'

import { Link } from 'react-router-dom'

const HomeView = () => (

  <section>

    <h1>Redux Tutorial</h1>

    <p>State management the classic way</p>

    <a href="https://redux.js.org/" target="blank">Read the docs</a>

    <hr />

    <Link to="/posts" className="button">

      Fetch Posts

    </Link>

  </section>

)

export default HomeView

**PostsView.jsx**

import React from 'react'

const PostsView = () => {

  return (

    <section>

      <h1>Posts</h1>

    </section>

  )

}

export default PostsView

Por cada posteo en la vista **PostsView** vamos a utilizar un componente **Post.jsx** que va a mostrar un resumen del texto de cada artículo. Este componente irá en la carpeta de componentes src/components/Post.jsx

import React from 'react'

const Post = ({ post }) => (

  <article className="post-excerpt">

    <h2>{post.title}</h2>

<p>{post.body.substring(0, 90)}...</p>

  </article>

)

export default Post

Sí, lo sé, no me digan nada… este componente no hace nada aún porque no lo estamos utilizando. Ya volveremos por él en un momento. Ahora vamos a introducir la funcionalidad del Router de React en el archivo **App.jsx**

import React from 'react'

import { BrowserRouter, Routes, Route } from 'react-router-dom';

import HomeView from './views/HomeView.';

import PostsView from './views/PostsView'

const App = () => {

  return (

    <BrowserRouter>

      <Routes>

        <Route path="/" element={<HomeView />} />

        <Route path="/posts" element={<PostsView />} />

      </Routes>

    </BrowserRouter>

  )

}

export default App

Ahora comienza la etapa de integración entre el componente **PostsView** y **Redux,** quevamos a conseguir con la función **connect** de **react-redux**. Connect es una función de orden superior (higher order function) que conecta el Store de Redux con un componente de React. La función **connect** espera un parámetro llamado **mapStateToProps**. Como bien indican la función y su parámetro, mapearemos cualquier estado del store de Redux a las props del componente. Le vamos a pasar ***loading***, ***posts*** y ***hasErrors***, que, como recordarán, están en el reducer (*postReducer.js)*.

src/views/PostsViews.jsx :

import React from 'react';

import { connect } from 'react-redux';

/\* El estado de Redux está a partir de aquí en

las props del componente \*/

const PostsView = ({ loading, posts, hasErrors }) => {

  return (

    <section>

      <h1>Posts</h1>

    </section>

  )

}

/\* Mapeamos el estado de Redux a las props

del componente \*/

const mapStateToProps = (state) => ({

  loading: state.posts.loading,

  posts: state.posts.posts,

  hasErrors: state.posts.hasErrors

})

//Y aquí conectamos Redux con React

export default connect(mapStateToProps)(PostsView);

Solo faltaría traernos a este módulo la función asíncrona **fetchPosts** que creamos antes en **postsActions.js**. Esa función es la acción que maneja todo el ciclo de vida de la petición HTTP en un solo lugar. La vamos a meter en el hook useEffect, que va a hacer el **dispatch** de **fetchPosts** cuando el componente se monte. El método **dispatch** va a estar disponible sin necesidad de importarlo siempre que lo usemos en un componente conectado a Redux a través de la función de orden superior **connect**. Obviamente, necesitamos pasarlo a las props del componente.

src/views/PostsViews.jsx :

import React, { useEffect } from 'react';

import { connect } from 'react-redux';

//Traemos la async action fetchPosts

import { fetchPosts } from '../actions/postsActions'

//Importamos el componente Post.jsx

import Post from '../components/Post';

/\* El estado de Redux está a partir de aquí en

las props del componente...

Y ahora incorporamos dispatch y useEffect \*/

const PostsView = ({ dispatch, loading, posts, hasErrors }) => {

  useEffect(() => {

    dispatch(fetchPosts())

  }, [dispatch])

  //Y hacemos una función para renderizar de acuerdo al resultado

  const renderPosts = () => {

    if (loading) return <p>Loading posts...</p>

    if (hasErrors) return <p>Something went South...</p>

    return posts && (posts.map((post) => <Post key={post.id} post={post} />))

  }

  return (

    <section>

      <h1>Posts</h1>

      {renderPosts()}

    </section>

  )

}

/\* Mapeamos el estado de Redux a las props del componente. Lo que escribimos aquí es con propósitos educativos. Serviría detallarlo en caso de que trajeramos piezas de estado de distintos reducers. En este caso de uso puntual, puede resumirse en una línea \*/

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

   loading: state.posts.loading,

   posts: state.posts.posts,

   hasErrors: state.posts.hasErrors

 }) \*/

const mapStateToProps = (state) => state.posts

//Y aquí conectamos Redux con React

export default connect(mapStateToProps)(PostsView);

Con lo visto hasta aquí tenemos un componente conectado (**PostView**) que trae datos desde una API y los almacena en el Redux store. Si abrimos las Redux DevTools podremos ver las actions mientras se van disparando y los cambios (diff) después de cada cambio. También podemos ir saltando de action en action y ver qué ocurría en cada momento.

Ahora solamente es cuestión de poner un poco de amor y dedicación a las técnicas que aprendimos para lograr algunos avances. Un menú y un componente para ver el detalle de cada post estarían muy bien.

Creación del componente de navegación

src/components/Navigator.jsx

import React from 'react'

import { NavLink } from 'react-router-dom'

const active = {

  border: '1px solid gray',

  color: 'darkgray'

}

const Navigator = () => (

  <nav>

    <section>

      <NavLink to="/"

        style={({ isActive }) => {

          return isActive ? active : null

        }}>Home</NavLink>

      <NavLink to="/posts"

        style={({ isActive }) => {

          return isActive ? active : null

        }}>Posts</NavLink>

    </section>

  </nav>

)

export default Navigator

Lo importaremos en el archivo App y vamos a insertarlo dentro de la estructura de <BrowserRouter>

import Navigator from './components/Navigator';

...

<BrowserRouter>

      <Navigator />

      <Routes>

...

Modificamos el componente Post.jsx

Queremos que el componente Post muestre el detalle de un artículo dado, al que accederemos por su id. Podríamos crear un nuevo componente, o reutilizar el que ya tenemos. Nos vamos a decantar por la segunda solución, así que le haremos unas modificaciones y ahora recibirá un booleano. Si está presente, renderizará el componente como un resumen del post y le agregará un botón ‘ver más’; si no, mostrará el texto completo del posteo sin el botón.

import React from 'react'

import { Link } from 'react-router-dom'

const Post = ({ post, excerpt }) => {

  return (

    <article className={excerpt ? "post-excerpt" : "post"}>

      <h2>{post.title}</h2>

      <p>{excerpt ? post.body.substring(0, 90) : post.body}...</p>

      {excerpt && (

        <Link to={`/posts/${post.id}`} className="button">Read More

        </Link>

      )}

    </article>

  )

}

export default Post

No olvidemos que en la vista **PostsView.jsx** necesitamos pasarle a **Post** la prop **excerpt** en la función **renderPosts().**

**src/views/PostsView**

const renderPosts = () => {

    if (loading) return <p>Loading posts...</p>

    if (hasErrors) return <p>Something went South...</p>

    return posts && (posts.map((post) => <Post key={post.id} post={post} excerpt/>))

  }

Si probamos nuestro programa tal y como está, veremos que levanta el id del posteo, pero no hemos configurado aún la ruta con parámetro ni el componente asociado, que, por otra parte, necesita ser creado. Será una vista del posteo elegido con todos los comentarios asociados que vienen de la API.

Primero resolvamos lo más sencillo, en **App.jsx:**

...

import SinglePostView from './views/SinglePostView';

...

<BrowserRouter>

      <Navigator />

      <Routes>

        <Route path="/" element={<HomeView />} />

        <Route>

          <Route path="/posts" element={<PostsView />} />

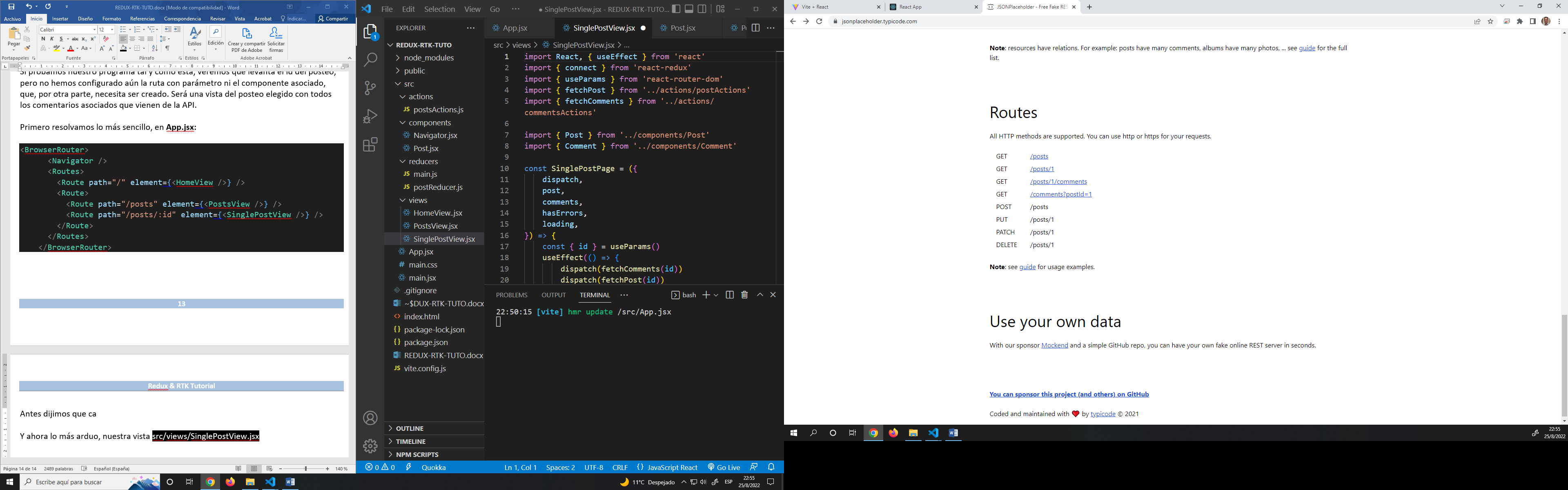
          <Route path="/posts/:id" element={<SinglePostView />} />

        </Route>

      </Routes>

    </BrowserRouter>

Si revisamos la documentación de jsonplaceholder.typicode.com que estamos usando como fuente de los datos, veremos que los posteos están relacionados a los comentarios.



En la necesidad de traer esos recursos es que vamos a crear un archivo de acciones, no ya para la serie de posteos sino para uno individual.

src/actions/postActions.js *(en singular):*

export const GET\_POST = 'GET POSTS'

export const GET\_POST\_SUCCESS = 'GET\_POST\_SUCCESS'

export const GET\_POST\_FAILURE = 'GET\_POST\_FAILURE'

export const getPost = () => ({ type: GET\_POST })

export const getPostSuccess = post => ({

  type: GET\_POST\_SUCCESS,

  payload: post,

})

export const getPostFailure = () => ({ type: GET\_POST\_FAILURE })

export function fetchPost(id) {

  return async dispatch => {

    dispatch(getPost())

    try {

      const res = await fetch(

        `https://jsonplaceholder.typicode.com/posts/${id}`)

      if (!res.ok) throw new Error(res.status)

      const data = await res.json()

      dispatch(getPostSuccess(data))

    } catch (error) {

      dispatch(getPostFailure())

    }

  }

}

Y su reducer. src/reducers/postReducer.js

import \* as actions from '../actions/postActions'

export const initialState = {

  loading: true,

  hasErrors: false,

  post: {},

}

export default function postReducer(state = initialState, action) {

  switch (action.type) {

    case actions.GET\_POST:

      return { ...state, loading: true }

    case actions.GET\_POST\_SUCCESS:

      return { post: action.payload, loading: false, hasErrors: false }

    case actions.GET\_POST\_FAILURE:

      return { ...state, loading: false, hasErrors: true }

    default:

      return state

  }

}

Si hemos visto que cada posteo está vinculado con varios comentarios, obviamente vamos a necesitar un archivo de actions para traerlos al store de Redux y poder usarlos cuando mostremos el posteo individual que los contiene.

src/actions/commentsActions.js

export const GET\_COMMENTS = 'GET COMMENTS'

export const GET\_COMMENTS\_SUCCESS = 'GET\_COMMENTS\_SUCCESS'

export const GET\_COMMENTS\_FAILURE = 'GET\_COMMENTS\_FAILURE'

export const getComments = () => ({ type: GET\_COMMENTS })

export const getCommentsSuccess = comments => ({

  type: GET\_COMMENTS\_SUCCESS,

  payload: comments,

})

export const getCommentsFailure = () => ({ type: GET\_COMMENTS\_FAILURE })

export function fetchComments(postId) {

  return async dispatch => {

    dispatch(getComments())

    try {

      const res = await fetch(

        `https://jsonplaceholder.typicode.com/comments?postId=${postId}`

      )

      if (!res.ok) throw new Error(res.status)

      const data = await res.json()

      dispatch(getCommentsSuccess(data))

    } catch (error) {

      dispatch(getCommentsFailure())

    }

  }

}

Y su reducer. src/reducers/commentsReducer.js

import \* as actions from '../actions/commentsActions'

export const initialState = {

  loading: false,

  hasErrors: false,

  comments: [],

}

export default function commentsReducer(state = initialState, action) {

  switch (action.type) {

    case actions.GET\_COMMENTS:

      return { ...state, loading: true }

    case actions.GET\_COMMENTS\_SUCCESS:

      return { ...state, comments: action.payload, loading: false, hasErrors: false }

    case actions.GET\_COMMENTS\_FAILURE:

      return { ...state, loading: false, hasErrors: true }

    default:

      return state

  }

}

El componente que renderiza cada comentario es muy simple:

src/components/Comment.jsx

import React from 'react'

const Comment = ({ comment }) => (

  <aside className="comment">

    <h2>{comment.title}</h2>

    <h3>{comment.email}</h3>

    <p>{comment.body}</p>

  </aside>

)

export default Comment

Finalmente, el componente que va a usar estas actions que creamos y el componente Comment, nuestra vista src/views/SinglePostView.jsx

import React, { useEffect } from 'react'

import { connect } from 'react-redux'

import { useParams } from 'react-router-dom'

import { fetchPost } from '../actions/postActions'

import { fetchComments } from '../actions/commentsActions'

import Post from '../components/Post'

import Comment from '../components/Comment'

const SinglePostPage = ({

  dispatch,

  post,

  comments,

  hasErrors,

  loading,

}) => {

  const { id } = useParams()

  useEffect(() => {

    dispatch(fetchComments(id))

    dispatch(fetchPost(id))

  }, [dispatch, id])

  const renderPost = () => {

    if (loading.post) return <p>Loading post...</p>

    if (hasErrors.post) return <p>Unable to display post.</p>

    return <Post post={post} />

  }

  const renderComments = () => {

    if (loading.comments) return <p>Loading comments...</p>

    if (hasErrors.comments) return <p>Unable to display comments.</p>

    return comments.map(comment => (

      <Comment key={comment.id} comment={comment} />

    ))

  }

  return (

    <section>

      {renderPost()}

      <h2>Comments</h2>

      {renderComments()}

    </section>

  )

}

const mapStateToProps = state => ({

  post: state.post.post,

  comments: state.comments.comments,

  loading: { post: state.post.loading, comments: state.comments.loading },

  hasErrors: { post: state.post.hasErrors, comments: state.comments.hasErrors },

})

export default connect(mapStateToProps)(SinglePostPage)

WTF? Por qué esto no arranca? ¿Habremos olvidado incorporar los nuevos reducers en el archivo **main.js de la carpeta reducers**?

import { combineReducers } from "redux";

import postsReducer from './postsReducer'

import postReducer from "./postReducer";

import commentsReducer from "./commentsReducer";

const rootReducer = combineReducers({

  posts: postsReducer,

  post: postReducer,

  comments: commentsReducer

});

export default rootReducer;