Xseed

Curso de React _ Technisys

Xseed

Clase 11 -

Redux Thunk

by Diego Cáceres

Repaso

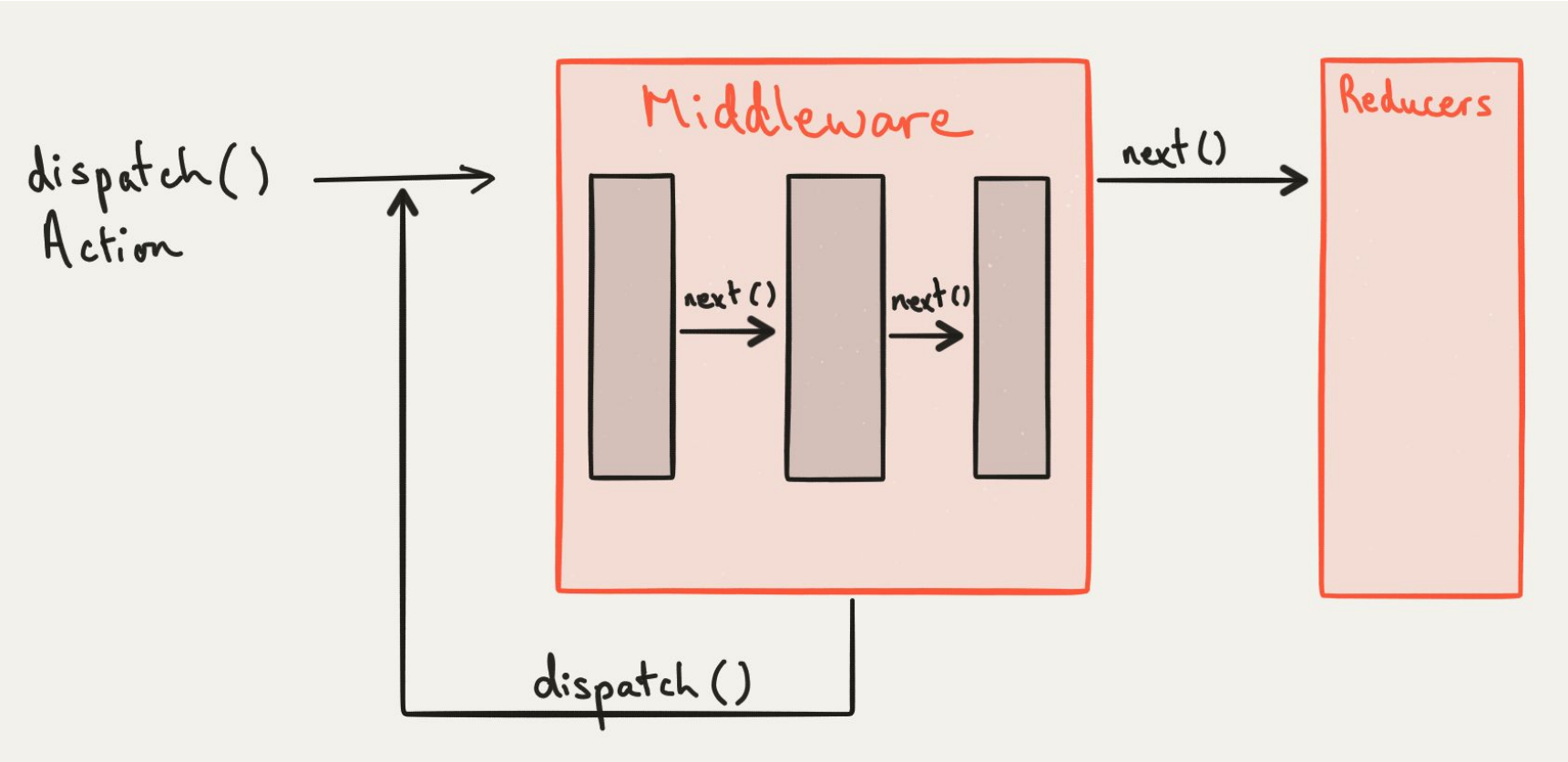
Redux Middleware

Los **Middlewares** son un punto de extensión en Redux, ya que nos proveen una forma de interactuar con todas las acciones que son emitidas al Reducer.

Hay varios ejemplos de uso, como loguear las acciones, reportar errores, hacer llamadas asíncronas, o dispatchear (emitir) nuevas acciones.

Los **Middlewares** se encadenan uno detrás de otro, y son llamados en orden para todas las acciones que se emitan. Tienen la posibilidad de modificar la Acción, o incluso de detener su propagación, osea, que no se envíe al siguiente **Middleware** ni al Reducer.

Redux Middleware



Redux Middleware - Implementación

Básicamente, es una función que recibe una acción e interactúa de alguna forma con ella:

```
function middleware(action) { /*...*/ }
```

Cómo funcionan en cadena, cada Middleware le debe enviar la acción al siguiente en la cadena, y si fuera el ultimo, seria al Reducer:

```
function middlewareWrapper(nextDispatch) {
 return function(action) {
     /*...*/
     nextDispatch(action);
 }
}
```

Redux Middleware - Implementación

Pero además, cada Middleware tiene acceso a una copia del Store (osea, tiene acceso a la función Dispatch y al getState para ver el State). Esto se logra siendo que la función final de un Middleware tiene otro wrapper:

```
function storeWrapper(store) {
 function middlewareWrapper(nextDispatch) {
   return function(action) {
     nextDispatch(action);
   }
 }
}
```

=

```
function middleware(store) {
 return function(nextDispatch) {
   return function(action) {
     nextDispatch(action);
   }
 }
}
```

=

```
const middleware = store => nextDispatch => action => {
 nextDispatch(action);
}
```

Redux Middleware

Para usar Middlewares se componen las distintas funciones utilizando un método de Redux llamado 'applyMiddleware', y dándole el resultado al createStore de Redux. En este caso, `middlewareOne` será el que reciba las acciones primero, y `middlewareTwo`, como es el último, es el que dispatcheara las acciones al Reducer.

```
import { applyMiddleware, createStore } from "redux";

const store = createStore(
 reducers,
 initialState,
 applyMiddleware(
   middlewareOne,
   middlewareTwo
 )
);
```

Side Effects

Side Effects – Redux (1)

Hasta ahora al emitir acciones de **Redux**, siempre son sincrónicas: inmediatamente luego de emitir una acción se actualizará el `state` del Store.

Pero en una aplicación lo normal es que determinadas acciones sean asincrónicas, por ejemplo, que consuman una API.

Hasta ahora, esto no es posible ya que los `reducers` son funciones puras (no pueden tener side effects) y las acciones son simples objetos JavaScript que describen lo que se quiere cambiar en el `state`.

Side Effects – Redux (2)

Cuando se consume una API **asincrónica**, normalmente hay dos eventos en el tiempo que interesan: cuando comienza la llamada, y cuando termina o devuelve timeout.

En el caso de Redux, esto se traduce en tres tipos de acciones que se podrían "dispatchear":

- Una acción que informa a los reducers que la request comenzó (esto podría por ejemplo cambiar una bandera que muestre el loader o spinner).
- Una acción que informa a los reducers que la request terminó satisfactoriamente (y pasarle los datos de la respuesta).
- Una acción que informa que la request falló (en este caso se puede mostrar un error o simplemente volver a intentarlo).

Side Effects – Redux (3)

Estas tres acciones se pueden definir con el mismo `type` pero utilizando, por ejemplo, una variable `status` adentro para diferenciarlas:

```
{ type: 'FETCH_POSTS' }
{ type: 'FETCH_POSTS', payload: { status: 'error', error: 'Oops' } }
{ type: 'FETCH_POSTS', payload: { status: 'success', response: 'respuesta' } }
```

O se pueden crear tres acciones con `type` distintos:

```
{ type: 'FETCH_POSTS_REQUEST' }
{ type: 'FETCH_POSTS_FAILURE', payload: { error: 'Oops' } }
{ type: 'FETCH_POSTS_SUCCESS', payload: { response: 'respuesta' } }
```

Lo más importante es que la decisión tomada se mantenga a lo largo de la aplicación.

Xseed

Redux Thunk

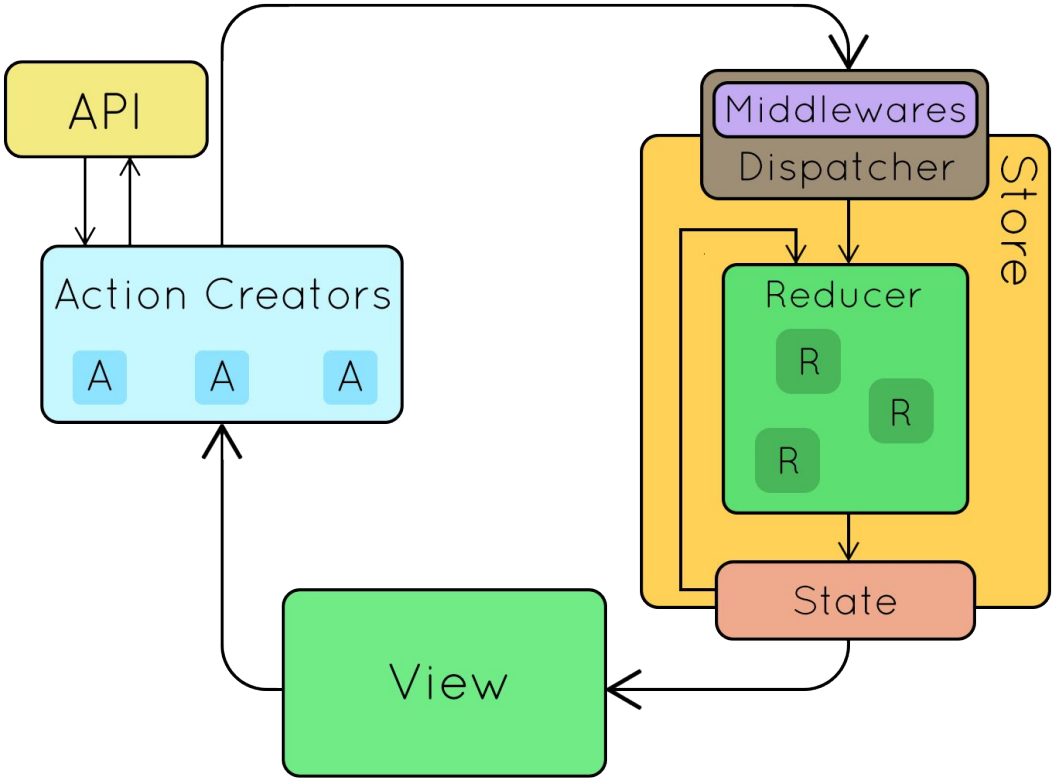
Redux Middleware – Redux Thunk

Para resolver los Side Effects en Redux, podemos utilizar un Middleware en particular llamado Redux Thunk Middleware.

Esta librería permite emitir un nuevo tipo de acciones, que son funciones, y se conocen como Thunks**.** El Middleware cuando detecta que es una función simplemente la invoca en vez de enviarla a los Reducers, y esta función tiene la posibilidad de "dispatchear" más acciones (ya sean ***thunks*** o acciones comunes).

Se crearán en el mismo lugar que las acciones hasta ahora, en un archivo llamado `actions.js` o similar.

Redux Middleware – Redux Thunk



```
export function fetchPosts() {
 // Thunk middleware se encarga de invocar la función con el dispatch.
 return function (dispatch) {
   // Dispatch para avisar que comienza la llamada a la API (loading).
   dispatch(requestPosts());
   return axios.get(`https://www.myapi.com/getPosts)
     .then(
       response => {
           console.log('Success getting posts.', response);
           dispatch(receivePosts(response.data));
       },
   // Pasamos una función para el error en vez de utilizar un catch
   // porque el catch agarra errores del dispatch tambien
       error => {
           console.log('An error occured.', error)
           dispatch(receivePostsFail(error));
       }
     );
 }
}
```

```
function requestPosts() {
 return {
     type: REQUEST_POSTS
 }
}

function receivePosts(posts) {
 return {
   type: RECEIVE_POSTS,
   payload: {
     posts: posts,
     receivedAt: Date.now()
   }
 }
}
```

Redux Middleware – Redux Thunk

La ventaja de utilizar este Middleware es que es transparente para los componentes.

En resumen, un **thunk** sigue siendo es una acción, pero antes de llegar al `reducer` se ejecuta su función y al `reducer` llegarán las acciones que esa función decida "dispatchear".

```
import { fetchPosts } from "../actions";
class ListPosts extends React.Component {
   componentDidMount(){
       this.props.fetchPostsLocal();
   }
   // render
}
const mapDispatchToProps = dispatch => {
   return {
       fetchPostsLocal: () => {
           return dispatch(fetchPosts());
       }
   };
};
export default
connect(null,mapDispatchToProps)(ListPosts);
```

Redux Middleware – Redux Thunk

La implementación de este Middleware es bastante simple, una versión simplificada del mismo sería así:

```
const thunkMiddleware = ({ dispatch, getState }) => next => action => {
 if (typeof action === 'function') {
   return action(dispatch, getState);
 }

 return next(action);
}
```

Redux Middleware – Redux Thunk

Para poder utilizar este Middleware con Redux (y cualquier otro) es necesario configurarlo al momento de crear el Store.

```
import thunkMiddleware from 'redux-thunk'
import { createStore, applyMiddleware } from 'redux'
import rootReducer from './reducers'
const store = createStore(
 rootReducer,
 applyMiddleware(
     thunkMiddleware // nos permite dispatchear funciones
 )
);
ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,  document.getElementById("root")
);
```

Además es necesario instalar la librería en la aplicación.

Ejercicios

Ejercicios

Descargar el zip `EjerciciosClase11`. Cada carpeta dentro corresponde a un ejercicio y es un proyecto creado con Create React App, por lo que para ejecutarlo es necesario:

- Si usan yarn:
  - `$ yarn install`
  - `$ yarn start`
- Si usan npm:
  - `$ npm install`
  - `$ npm start`

Se debería abrir automáticamente el explorador con el ejercicio corriendo en http://localhost:3000/

Ejercicio 1

Este ejercicio es la aplicación de **ToDos** (tareas pendientes). Ya tiene redux para el manejo del estado.

- Existe un nuevo componente en `src/FilterTodos` que tiene tres links. Se necesita conectar este componente al Store de Redux para que lea el filtro actual y lo muestre, y además pueda cambiar el filtro al hacer click en los links (va a "dispatchear" la acción correspondiente).
    - Los valores que manda en la acción para el parámetro **filter** serán: "all", "completed" y "active".
- Luego, agregar al proyecto redux-thunk.
  ```
  $ npm install --save "redux-thunk"
  ```
- Configurar el Middleware en `index.js`.

Ejercicio 1 (cont.)

Hay creado un servidor falso en el archivo `api.js`.

Es necesario implementar un **thunk** llamado `getTodosFiltered`, una acción de tipo función para que consuma la API `fetchTodos` y como respuesta se guarden los **ToDos** en el Store.

- Primero, se deben crear los tres action creators (`fetchTodosStart`, `fetchTodosSuccess`, `fetchTodosFail`) utilizando los types ya definidos.
- Luego, crear este thunk que consuma `fetchTodos` de `api.js` y "dispatchee" las acciones correspondientes.
- Implementar el reducer para que guarde el array de `ToDos` en el Store para el action type `FETCH_TODOS_SUCCESS`.

Ejercicio 1 (cont.)

- Agregar en TodoList para que reciba del Store tambien el `visibilityFilter`.
- En TodoList implementar que en el método `componentDidMount` "dispatchee" la acción/thunk `getTodosFiltered` (importarla de `actions.js`) con el `visibilityFilter` correspondiente.
- Notar que si cambia el `visibilityFilter` TodoList no está volviendo a "dispatchear" la acción `getTodosFiltered`, por lo que es necesario implementarlo:
    - Implementar el método `componentDidUpdate` y comparar si el `visibilityFilter` cambió, para volver a actualizar los **ToDos**.

```
componentDidUpdate(prevProps, prevState) {
    if (prevProps.visibilityFilter !== this.props.visibilityFilter) {
        this.props.getTodosFilteredLocal(this.props.visibilityFilter);
    }
}
```

Ejercicio 1 (cont.)

- Se debe agregar un nuevo valor en el `state` del Store: una bandera llamada `loading` que por defecto es `false`.
- Modificar `TodoList` para que acceda a este valor desde Redux y si está cargando no muestre la lista, sino muestre un mensaje de *"Loading..."*.
- Agregar en el reducer que el action type `FETCH_TODOS_START` ponga en `true` la bandera y `FETCH_TODOS_SUCCESS` o `FETCH_TODOS_FAIL` la pongan en `false`.

```
if (action.type === types.FETCH_TODOS_START) { // Dentro del reducer.
    return {
        ...state,
        loading: true
    };
}
```

Ejercicio 2

En este ejercicio hay una lista harcodeada de usuarios que se está mostrando.

- Consumir desde la api “https://jsonplaceholder.typicode.com/users” una lista de usuarios y guardarla en Redux, de forma de dejar de mostrar los usuarios harcodeados, y mostrar la respuesta de la lista.
- Luego, al hacer click en un usuario de la lista, queremos que se despliegue el usuario seleccionado, osea que hay que disparar una acción que lo guarde en Redux, y conectar el componente correspondiente.
- Tambien queremos ver sus posts, que se obtienen desde “https://jsonplaceholder.typicode.com/posts?userId=2” sustituyendo el Id del usuario clickeado. Osea que luego de seleccionarlo, hay que ir a buscar esta info y dejarla en el Store de Redux, para conectar el componente correspondiente.