Curso de React y React Native

Clase 07

Agenda de la clase

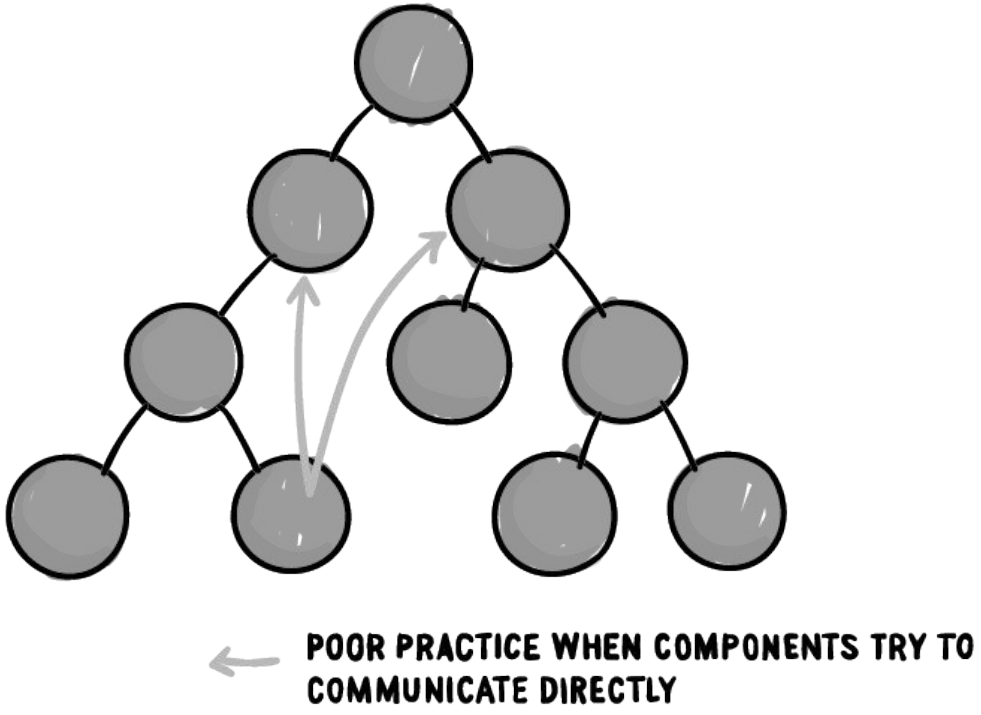
Agenda

- Repaso.
- Side Effects
  - Redux Middleware
  - Redux Thunk
- Ejercicios.

Repaso

REPASO

React – Problema con el flujo de datos



REPASO

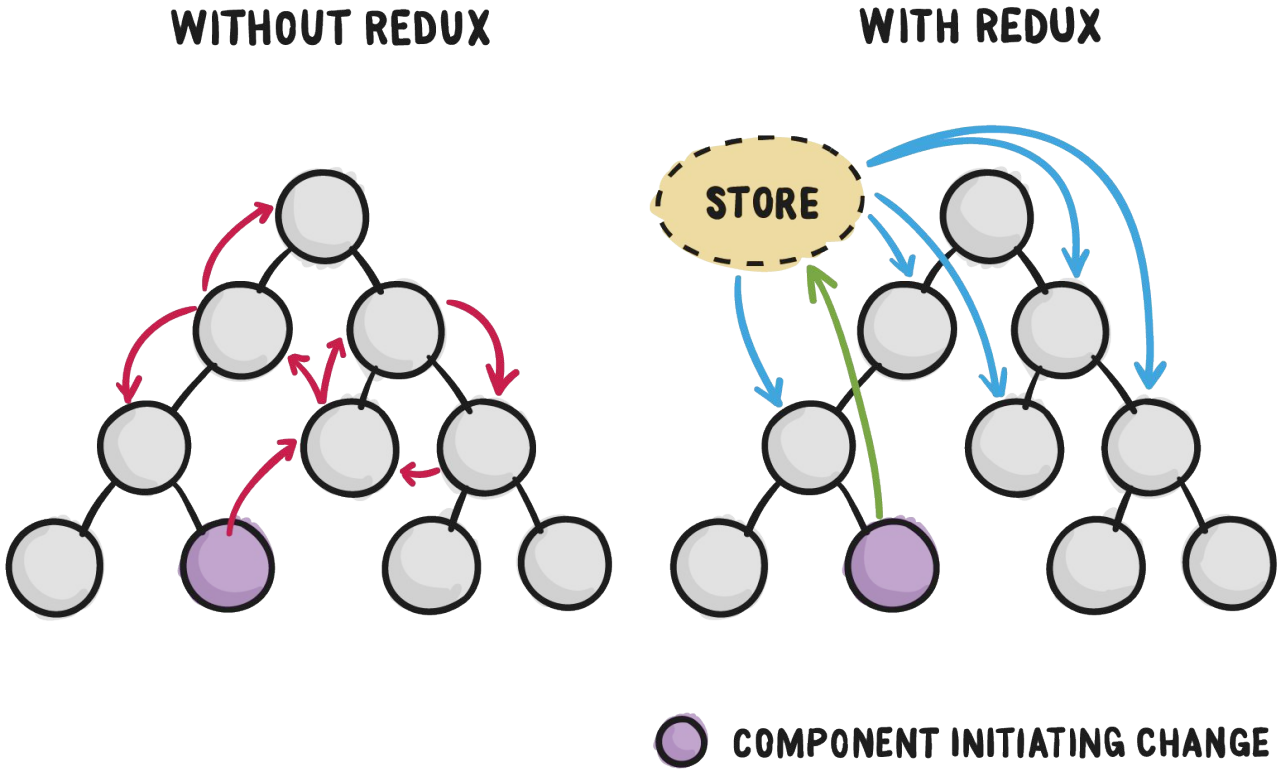
React – Problema con el flujo de datos

La solución que brinda Redux para esto es simple:

Una estructura única que guarda todo el estado de la aplicación en tiempo de ejecución, llamada Store.

REPASO

React – Redux



REPASO

Redux – Store's state

El Store es donde se guarda el `state` de la app entera, en un árbol de objetos.

```
todos: [
   { id: 1, name: 'LearnReact', isComplete: false },
   { id: 2, name: 'Learn Redux', isComplete: true },
   { id: 3, name: 'Learn ReactNative ', isComplete: false },
   { id: 4, name: 'Learn NodeJS', isComplete: false }
]
```

Un detalle no menor es que el Store es **Read Only**, la forma de modificarlo es emitiendo acciones, llamadas Actions.

REPASO

Redux – Actions

- Las acciones son simples objetos JavaScript.
- Tienen un **type** (obligatorio).
- Tienen un **payload**, que puede ser cualquier cosa (opcional).
- Su propósito es describir un evento.
- El evento puede provocar un cambio en el state -> Mediante un **Reducer.**

```
const action = {
    type: "Type of the action",
    payload: { data: "Information" }
}
```

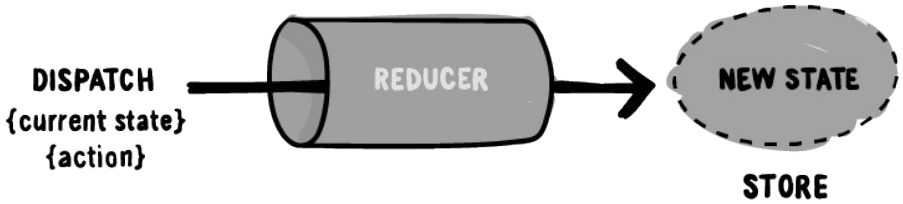
```
const addTodo = {
    type: "ADD_TODO",
    payload: "Learn Redux"
}
```

REPASO

Redux – Reducers (1)

Los **Reducers** son funciones JavaScript con la condición de ser funciones **puras**. No modifican los parámetros que reciben y no dependen de nada externo a la función (dado un mismo *input* siempre producen el mismo *output*).

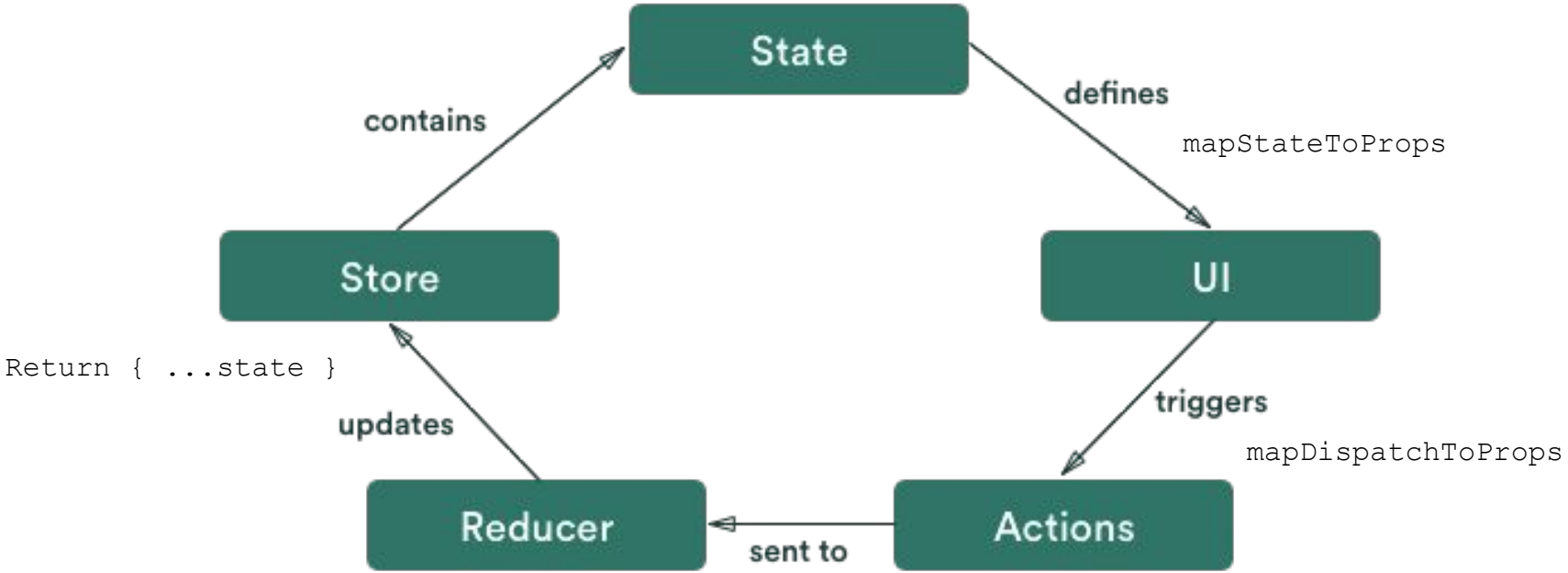
- Reciben el state actual,
  y el action que ocurrió.
- Siempre devuelven un nuevo state




```
function reducer(state, action) {
   // Nos aseguramos que este definido:
   state = state || initialState;
   if(action.type === "type1"){
       // Calcular nuevo estado.
   }
   else if(action.type === "type2"){
       // Calcular nuevo estado.
   }
   return state;
}
```

REPASO

Redux Flow



REPASO

React-Redux

React-redux es una librería que provee bindings para facilitar el uso de Redux en React.

- `Provider` => inyecta el store en React.
- `connect` => es un HOC que permite conectar cualquier componente en la jerarquía al Store de Redux (siempre y cuando esté dentro de Provider).

```
import { Provider } from 'react-redux';
import reducer from './reducer'

let store = createStore(reducer)
ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

REPASO

React-Redux

`mapStateToProps` es una función que recibe el *state* del Store. Las propiedades del objeto que retorna a partir del *state* llegarán al componente que se conecte como `props` (`TodoList` en el ejemplo). Algo similar sucede con `mapDispatchToProps`, pero aquí se definen las funciones que se quiere que el componente que se conecte reciba por props, de forma de poder "dispatchear" acciones a los Reducers.

```
const mapStateToProps = (state, [ownProps]) => ({
    todos: state.todos
})
const mapDispatchToProps = (dispatch, [ownProps]) => ({
    addTodo: text => dispatch(addTodo(text))
})
export default connect(
    mapStateToProps,
    mapDispatchToProps
)(TodoList);
```

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 es de forma **asincrónica**, y normalmente hay dos eventos en el tiempo que interesan: cuando comienza la llamada y cuando termina (o responde 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.

Redux Middleware

Es necesario poder emitir acciones que tengan Side Effects, por ejemplo, para ir a buscar a una API los Posts (Artículos) que luego la aplicación va a mostrar.

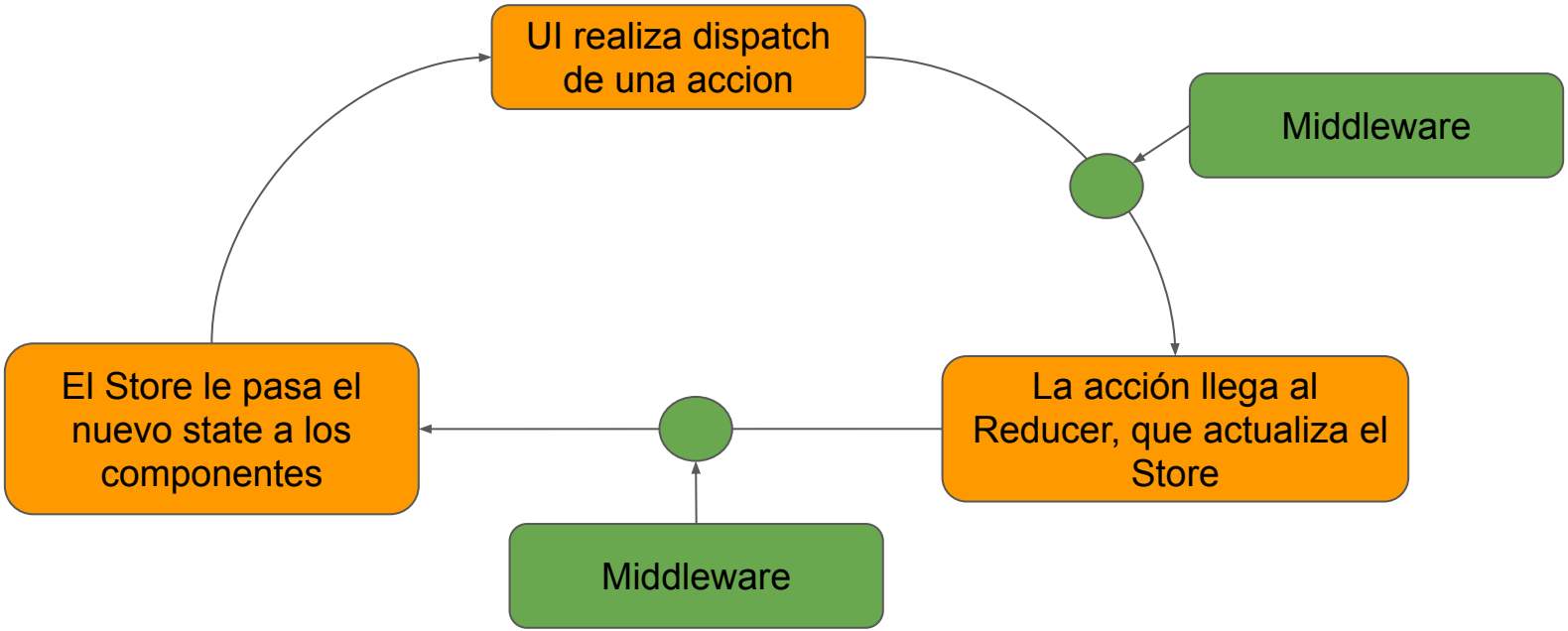
Para esto, se va a utilizar el concepto de Middleware, particularmente en **Redux Middleware**.

Un Middleware es, por definición, un software utilizado para ayudar a la conexión entre dos aplicaciones, algo que se ejecuta "en el medio".

En Redux se pueden utilizar varios Middlewares distintos y tienen dos lugares donde se pueden ejecutar:

- Entre que se "dispatchea" una acción y llega al reducer.
- Inmediatamente luego de que se ejecuta el reducer, con el state actualizado.

Redux Middleware



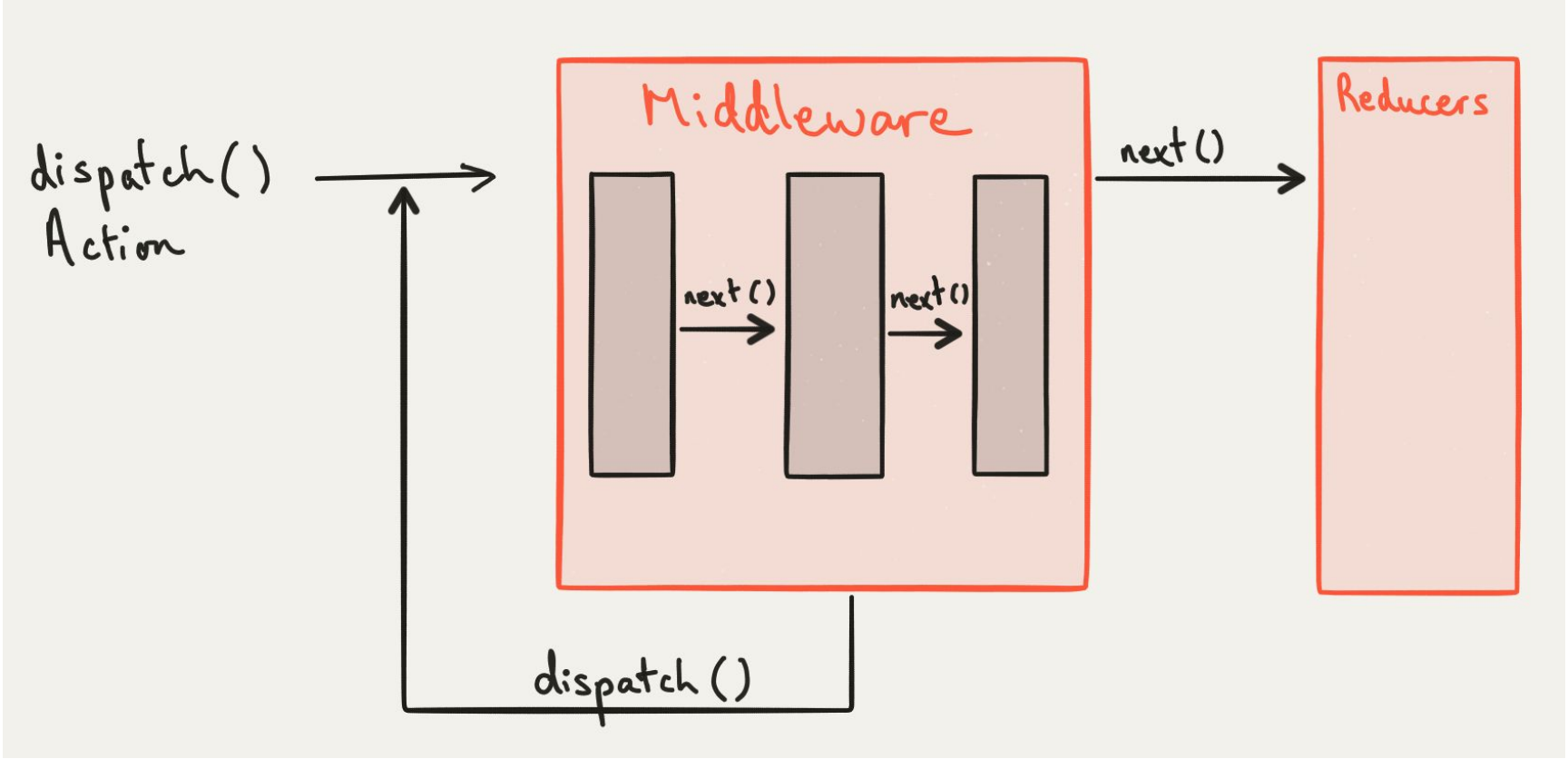
Redux Middleware

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

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
  )
);
```

Redux Middleware – Redux Thunk

Esto nos da la posibilidad de utilizar un Middleware en particular llamado Redux Thunk Middleware.

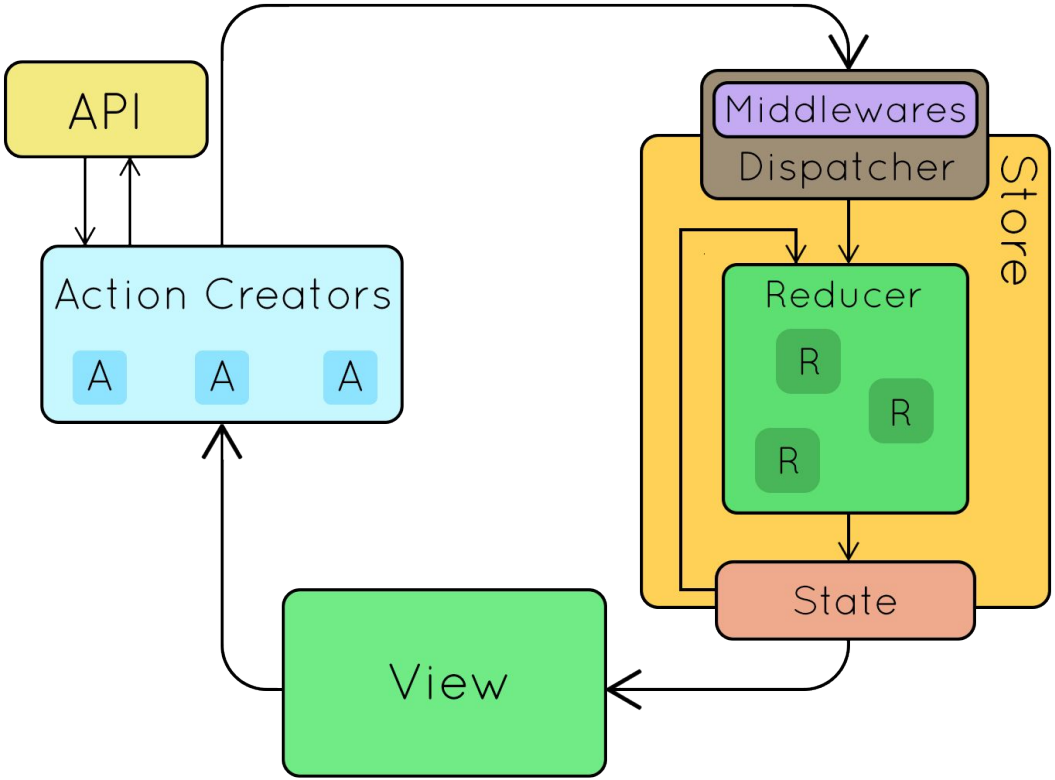
Esta librería permite despachar un nuevo tipo de acciones. En lugar de ser objetos, son funciones, y se conocen como Thunks**.**

Cuando el Middleware detecta que la acción es una función, la invoca.

Esta función tiene la posibilidad entonces de despachar más acciones (ya sean más ***thunks*** –es decir, funciones– o acciones comunes –Objetos de JavaScript–).

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

Redux Middleware



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));
       },
       // Es mejor pasarle una función para el error que utilizar un catch
       // sino el catch agarra errores del dispatch y puede quedar en loop.
       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.

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

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),
);

ReactDOM.render(
   <Provider store={store}>
     <App />
   </Provider>,  document.getElementById("root")
);
```

Además es necesario instalar la librería en la aplicación.

Redux Middleware – Redux Thunk

```
import thunkMiddleware from 'redux-thunk'
import { createLogger } from 'redux-logger'
import { createStore, applyMiddleware } from 'redux'
import rootReducer from './reducers'

const loggerMiddleware = createLogger();
const store = createStore(
      rootReducer,
      applyMiddleware(
            thunkMiddleware, // nos permite dispatchear funciones
            loggerMiddleware // middleware que loguea toda las acciones
      )
);
ReactDOM.render(
   <Provider store={store}>
     <App />
   </Provider>, document.getElementById("root")
);
```

`applyMiddleware` puede recibir más de un Middleware.

Ejercicios

Ejercicio

1. Crear un proyecto de React.
2. Instalar `redux`, `react-redux`, `redux-thunk` y `redux-logger`.
3. Utilizando la API de CoinDesk, mostrar el precio actual de Bitcoin cuando carga la página, ofreciendo a la vez un botón de actualizar, el cual al ser clickeado “refresque” dicho valor.
4. Mientras se realiza la request mostrar un spinner o similar (“...” por ejemplo).

La idea del ejercicio, naturalmente, es utilizar redux y redux-thunk para lograr los resultados esperados.

Para quienes dominen async/await, buenas noticias: redux-thunk funciona perfectamente con esta sintáxis.