

Tema 5

React y Redux en el mundo real

La teoría



La práctica



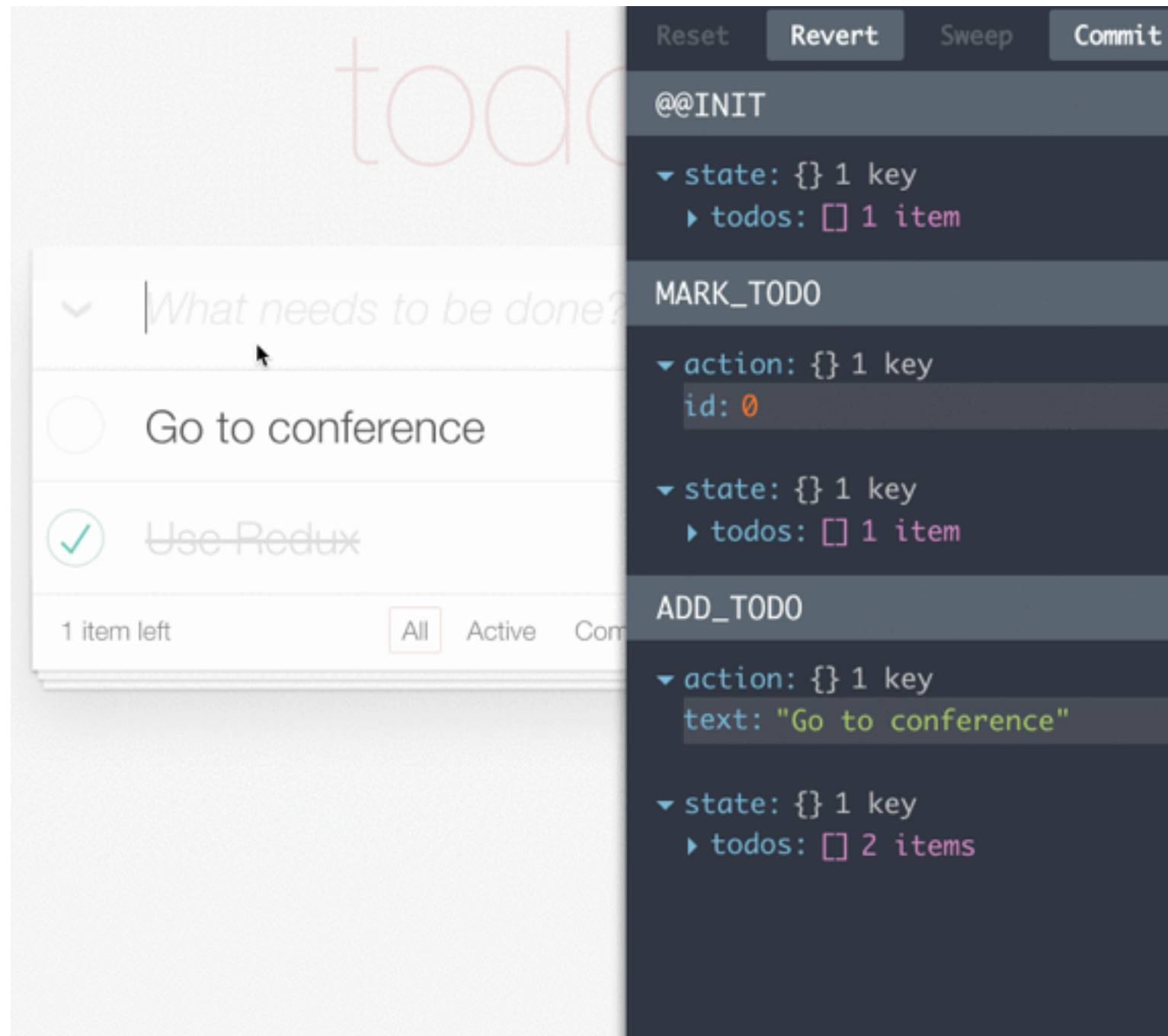
Imprescindibles

- Experiencia de desarrollo con Redux Dev Tools
- Routing
- Acceso a APIs REST

Redux Dev Tools

- Una herramienta de ayuda al desarrollo para monitorizar el Store de Redux
- Igual que podemos examinar el árbol de componentes con **React Dev Tools**
- Podremos examinar el árbol de estado, y ver las acciones que se despachan, con **Redux Dev Tools**

Redux Dev Tools



Redux Dev Tools

- Dos versiones: una que integramos **manualmente** en nuestro código
- Otra que es una extensión de Chrome (como React Dev Tools) y que necesita mucha menos configuración
- La primera permite mucha más personalización

Redux Dev Tools - manual

- La herramienta se compone de dos partes:
 - Una que se inyecta en el Store y que permite registrar todo lo que ocurre (un *store enhancer*)
 - Una serie de componentes para visualizar todo esto en paralelo a nuestra aplicación

Redux Dev Tools - manual

- Instalamos como dependencia **de desarrollo**:
- **npm install -D redux-devtools**
- Hay diferentes "monitores" (UI) para escoger, vamos a instalar lo básico:
- **npm install -D redux-devtools-dock-monitor**
- **npm install -D redux-devtools-log-monitor**

Redux Dev Tools - manual

- Nos creamos un componente **DevTools** con el siguiente código (copy paste!!!)
- Lo que hace es simplemente importar los monitores y conectarlos con el store

Redux Dev Tools - manual

```
import React from 'react';

// Exported from redux-devtools
import { createDevTools } from 'redux-devtools';

// Monitors are separate packages, and you can make a custom one
import LogMonitor from 'redux-devtools-log-monitor';
import DockMonitor from 'redux-devtools-dock-monitor';

// createDevTools takes a monitor and produces a DevTools component
const DevTools = createDevTools(
  // Monitors are individually adjustable with props.
  // Consult their repositories to learn about those props.
  // Here, we put LogMonitor inside a DockMonitor.
  // Note: DockMonitor is visible by default.
  <DockMonitor toggleVisibilityKey='ctrl-h'
               changePositionKey='ctrl-q'
               defaultIsVisible={true}>
    <LogMonitor theme='tomorrow' />
  </DockMonitor>
);

export default DevTools;
```

Redux Dev Tools - manual

- Para conectarlo con el store, necesitamos **dos** *store enhancers*: el **applyMiddleware** que teníamos y uno nuevo que obtenemos del componente que hemos creado
- Podemos componer ambos con **compose**

Redux Dev Tools - manual

configureStore.js

```
import {  
  createStore, combineReducers,  
  applyMiddleware, compose } from 'redux';  
import DevTools from './components/devTools';  
import reducers from './modules/reducers';  
import thunk from 'redux-thunk';  
  
const appReducer = combineReducers(reducers);  
const enhancer = compose(applyMiddleware(thunk), DevTools.instrument());  
  
export default function configureStore() {  
  return createStore(appReducer, enhancer);  
}
```

Redux Dev Tools - manual

src/modules/reducers.js

```
import cart from './cart';  
import catalog from './catalog';  
import order from './order';  
import route from './route';  
  
export default {  
  cart,  
  catalog,  
  order,  
  route  
};
```

Redux Dev Tools - manual

- Ya tenemos nuestro Store "tuneado"
- Sólo nos falta incluir el componente **DevTools** dentro de nuestra app
- Normalmente lo incluiremos en el **render** de nuestro componente raíz

Redux Dev Tools - manual



src/components/ecommerce/index.js

```
import React, { Component, PropTypes } from 'react';
import { connect } from 'react-redux';
//DevTools
import DevTools from '../devTools';

//....
render() {
  const component = this.getComponentForPage(this.props.page);
  return (
    <div className='shopping-cart'>
      { component }
      <DevTools />
    </div>
  )
}
```


Redux Dev Tools - manual

Products

iPhone 6 	Grandote pero chulo	699.99 € Comprar
Nexus 7 	El mejor Android de la historia.	799.99 € Comprar
Feberphone 	Dale a tu nene el Feber, no te vaya a romper el de verdad.	899.99 € Comprar
Woodyphone 	En madera reciclada. Cero potencia, todo conciencia.	899.99 € Comprar

Reset Revert Sweep Commit

@@@INIT

▼ state: {} 4 keys

▶ cart: {} 0 keys

▶ catalog: [] 0 items

▶ order: {} 2 keys

route: "catalog"

SAVE_CATALOG

▼ action: {} 1 key

▶ payload: [] 4 items

▼ state: {} 4 keys

▶ cart: {} 0 keys

▶ catalog: [] 4 items

▶ order: {} 2 keys

route: "catalog"

Redux Dev Tools - manual

- Hay muchos otros "monitores" que podemos usar en lugar de **LogMonitor**
- Lista actualizada en <https://github.com/gaearon/redux-devtools#custom-monitors>
- Además cada monitor tiene su propia API (props) para alterar su funcionamiento (filtrar acciones, mostrar sólo una parte del state, etc).

Redux Dev Tools - extensión de Chrome

- <https://github.com/zalmoxisus/redux-devtools-extension>
- Disponible en **Chrome Store** para instalar desde Chrome
- Sólo hay que escribir una línea en **configureStore.js**, que detectará si está o no instalada la extensión
- Y no tenemos que incluir `<DevTools />` en el render, ya que está disponible en el propio Chrome

Redux Dev Tools - extensión de Chrome

configureStore.js

```
import { createStore, combineReducers, applyMiddleware, compose } from 'redux';  
import reducers from './modules/reducers';  
import thunk from 'redux-thunk';  
  
const appReducer = combineReducers(reducers);  
//devtools extension  
const enhancer = compose(  
  applyMiddleware(thunk),  
  window.devToolsExtension ? window.devToolsExtension() : f => f);  
  
export default function configureStore() {  
  return createStore(appReducer, enhancer);  
}
```

Redux Dev Tools

- No son la panacea, pero ofrecen una ayuda al desarrollo en tiempo real
 - Cargar / descargar el estado en formato JSON
 - Hacer / deshacer acciones concretas
 - Despachar acciones "a mano"
 - ...

Routing

- Nuestro Ecommerce tiene un fallo claro...
- El usuario no puede navegar adelante/atrás por nuestras páginas
- Al construir una SPA, no queremos romper los hábitos del usuario
- Las URLs no pueden compartirse ni añadirse a marcadores

Routing en React

- React no incluye ninguna solución de routing
- Puedes emplear el que quieras...
Backbone.Router, Page, Director
- Pero en realidad, ¿para qué usamos el router?

Routing en React

- Pintar un componente/layout por ruta
- Ejecutar código cuando haya coincidencia de rutas (ej: cargar datos, etc.)

React-router

<https://github.com/reactjs/react-router>



REACT/ROUTER

React-router

- Las rutas se definen como componentes React
- ¿ Rutas == UI ?
- Igual que JSX: “dale cinco minutos”
- **npm install --save react-router**

React-router

```
import { Router, Route, IndexRoute, browserHistory } from  
  'react-router';
```

```
const ExampleRouter extends Component {  
  render() {  
    return (  
      <Router history={ browserHistory }>  
        <Route path="/" component={ Layout }>  
          <IndexRoute component={ Home } />  
          <Route path="about" component={ About } />  
          <Route onEnter={ loadContact }  
            path="contact/:id" component={ Contact } />  
        </Route>  
      </Router>  
    )  
  }  
}
```

React-router

```
import { Router, Route, IndexRoute, browserHistory } from  
  'react-router';
```

```
const ExampleRouter extends Component {  
  render() {  
    return (  
      ➔ <Router history={ browserHistory }>  
        <Route path="/" component={ Layout }>  
          <IndexRoute component={ Home } />  
          <Route path="about" component={ About } />  
          <Route onEnter={ loadContact }  
            path="contact/:id" component={ Contact } />  
        </Route>  
      </Router>  
    )  
  }  
}
```

El **Router** es nuestro componente de más alto nivel

React-router

```
import { Router, Route, IndexRoute, browserHistory } from  
  'react-router';
```

```
const ExampleRouter extends Component {  
  render() {  
    return (  
      <Router history={ browserHistory }>  
        ➔ <Route path="/" component={ Layout }>  
          <IndexRoute component={ Home } />  
          <Route path="about" component={ About } />  
          <Route onEnter={ loadContact }  
            path="contact/:id" component={ Contact } />  
        </Route>  
      </Router>  
    )  
  }  
}
```

Sus hijos son rutas individuales
Definen para un **path** dado, qué
componente debe mostrarse

React-router

```
//routes
```

```
import ExampleRouter from './routes/example';
```

```
window.onload = function() {
```

```
  ReactDOM.render(<ExampleRouter />, document.getElementById('app'));  
}
```

Ventajas React-Router

- Rutas anidadas nos generan UI con componentes anidados
- La configuración de rutas tiene la misma **forma** que la UI que queremos generar
- Posibilidad de ejecutarlo en el servidor (server-side rendering)
- Compatible con #hash y HTML5 history

Configurar history

- React-router exporta dos tipos de "history" como singleton: **hashHistory** y **browserHistory** (HTML5)
- Debemos escoger uno e inyectarlo como prop "history" al Router en sí

Configuración de rutas

- Componente **Route** acepta como props obligatorios:
 - **path** - Ruta con parámetros al estilo habitual
 - **component** - Componente React que debe mostrarse para esa ruta
- Opcionales:
 - **onEnter** - función de transición de entrada
 - **onLeave** - función de transición de salida

Configuración de rutas

- **onEnter** espera una función la siguiente firma

```
function (nextState, replaceState)
```

- Donde nextState contiene el “estado” del Router, por ejemplo los parámetros de ruta en **nextState.params** y replaceState es una función para reescribir la ruta
- O si queremos que nuestro código sea asíncrono:

```
function(nextState, replaceState, done)
```

- Cuando queramos que se complete la transición, llamaremos a **done**

Configuración de rutas

- Los componentes **Route** se pueden anidar para crear rutas anidadas complejas y UIs complejas
- Si la ruta tiene parámetros, el componente los recibe como props, dentro de la clave **params**
- El componente debe incluir en su **render** `{ props.children }` para pintar los componentes anidados si la ruta los acepta

Sintaxis para path

- **:parametro** - Concide con un segmento en la URL hasta el siguiente separador (/, #, ?). Ese valor lo tendremos en **props.params.parametro** en el Route Component
- **(x)** - Opcional
- ***** - Coincide con cualquier carácter (non-greedy)
- ****** - Coincide con cualquier carácter (greedy)

Sintaxis para path

// matches **/hello/michael** and **/hello/ryan**

```
<Route path="/hello/:name">
```

// matches **/hello**, **/hello/michael**, and **/hello/ryan**

```
<Route path="/hello(/:name)">
```

// matches **/files/hello.jpg** and **/files/hello.html**

```
<Route path="/files/*.*)">
```

// matches **/files/hello.jpg** and **/files/path/to/file.jpg**

```
<Route path="/**/*.jpg">
```

Anidar rutas

- React Router nos permite declarar conjuntos de vistas anidadas que queremos mostrar al entrar en una URL
- Al buscar coincidencias, React Router busca en profundidad, de dentro a fuera

Anidar rutas - ejemplo

- Supongamos una aplicación de gestión de contactos:
- `/` - home de la aplicación
- **`/contacts`** - listado de contactos
- **`/contacts/:id`** - detalle de un contacto
- **`/contacts/:id/edit`** - formulario de edición de un contacto
- Y además queremos ir anidando los "layout" unos dentro de otros...

Anidar rutas - ejemplo

```
<Router history={ history }>
  <Route path="/" component={Layout}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ ContactsLayout }>
      <IndexRoute onEnter={ loadAllContacts } component={ ContactList } />
      <Route onEnter={ loadContact } path=":id" component={ ContactDetails } />
      <Route onEnter={ loadContact } path=":id/edit" component={ ContactForm } />
    </Route>
  </Route>
</Router>
```


Anidar rutas

```
<Router history={ history }>
  <Route path="/" component={Layout}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ ContactsLayout }>
      <IndexRoute onEnter={ loadAllContacts } component={ ContactList } />
      <Route onEnter={ loadContact } path=":id" component={ ContactDetails } />
      <Route onEnter={ loadContact } path=":id/edit" component={ ContactForm } />
    </Route>
  </Route>
</Router>
```

Si la ruta es "/" se pintará Home dentro de Layout

Anidar rutas

```
<Router history={ history }>
  <Route path="/" component={Layout}>
    <IndexRoute component={Home} />
    <Route path="contacts" component={ ContactsLayout }>
      <IndexRoute onEnter={ loadAllContacts } component={ ContactList } />
      <Route onEnter={ loadContact } path=":id" component={ ContactDetails } />
      <Route onEnter={ loadContact } path=":id/edit" component={ ContactForm } />
    </Route>
  </Route>
</Router>
```

Si la ruta es “/contacts” se pintará ContactList dentro de ContactsLayout, y éste dentro de Layout

Anidar rutas

```
<Router history={ history }>
  <Route path="/" component={ Layout }>
    <IndexRoute component={ Home } />
    <Route path="contacts" component={ ContactsLayout }>
      <IndexRoute onEnter={ loadAllContacts } component={ ContactList } />
      <Route onEnter={ loadContact } path=":id" component={ ContactDetails } />
      <Route onEnter={ loadContact } path=":id/edit" component={ ContactForm } />
    </Route>
  </Route>
</Router>
```

Si la ruta es “/contacts/256” se pintará
ContactDetails dentro de
ContactsLayout, dentro a su vez de
Layout

Anidar rutas

```
<Router history={ history }>
  <Route path="/" component={ Layout }>
    <IndexRoute component={ Home } />
    <Route path="contacts" component={ ContactsLayout }>
      <IndexRoute onEnter={ loadAllContacts } component={ ContactList } />
      <Route onEnter={ loadContact } path=":id" component={ ContactDetails } />
      <Route onEnter={ loadContact } path=":id/edit" component={ ContactForm } />
    </Route>
  </Route>
</Router>
```

Si la ruta es “/contacts/256/edit” se pintará ContactForm dentro de ContactsLayout, dentro a su vez de Layout

Anidar rutas - ejemplo

- Los componentes tienen acceso a los hijos con **this.props.children**
- Así es como anidamos unos dentro de otros

<Route> en detalle (propiedades)

- path (String) - Ej: **/cart**, **/product/:id**
- component (React element) - Ej: <Layout />
- onEnter (function(nextState, replace))
 - nextState tiene **params**, location, etc
 - replace es una función con la que podemos redirigir
- onLeave(function())

Ejercicio - añadir Router a nuestra aplicación

- Para usar **browserHistory** (sin hash), el servidor tiene que ser compatible
- Podemos añadir **historyApiFallback: true** a `webpack.config.js`, dentro de **devServer** (ver `ejercicios/tema5/src/webpack.config.js`)
- Esto hará que `webpack-dev-server` nos devuelva siempre `index.html` para cualquier ruta (`/cart`, etc)

Ejercicio - añadir Router a nuestra aplicación

- Usamos un componente **layout.js** que incluya el HTML actual de index.js y que pinte los "hijos" que le inyecte el Router
- Index.js ahora será nuestro Router
- Las rutas se pueden definir en un archivo separado **routes.js**

/ejercicios/tema5/src/components/ecommerce/routes.js

- El esqueleto ya incluye el Router configurado

Ejercicio - añadir Router a nuestra aplicación

- ¿Y ahora cómo navegamos al carrito al añadir un producto?
- Varias alternativas
 - Componentes **<Link />**, **<IndexLink />**
 - Decorador **withRouter**

Navegar desde JSX con Link

- La librería ofrece componentes **Link**, **IndexLink** que generan por nosotros el hiperenlace apropiado
- Lo podemos usar en lugar de `<a href ...>` y detecta si está "activo" automáticamente

```
<Link to='/home' activeClassName='active'>Texto</Link>  
<IndexLink to='/home/about' activeClassName='active'>  
  Texto</IndexLink>
```

Navegar desde JSX con Link

- Podemos añadirle además **onChange** y manejar (y cancelar) el evento de navegación
- También podemos props que queramos que tenga el enlace, como **title, id, className...**
- Es la forma más habitual de navegar, especialmente desde menús

Navegar desde código con `withRouter()`

- La librería ofrece un decorador, **withRouter**, para envolver cualquier componente en el árbol del router (al estilo `connect()` de Redux)
- Nos inyectará la prop **router** al componente
- Y ese router ofrece métodos para manipular el histórico: **push(path)**, **replace(path)**, **go(n)**, **goBack()**, **goForward()**

Navegar desde código con withRouter()

```
import React from 'react';
import Header from './header';
➔ import { withRouter } from 'react-router';

// Página de "No encontrado"
const NotFound = (props) => {
  const { router } = props;
  return (
    <div className='not-found'>
      <Header text='Page not found' />
      <p>Ooops! Not ready yet</p>
      <p>
        <a className='button' onClick={ () => { router.push('/') } }>
          Back to Shop
        </a>
      </p>
    </div>
  );
}

➔ export default withRouter(NotFound);
```

Ejercicio - añadir Router a nuestra aplicación

- Usando **withRouter** cambiad el componente Catalog para que navegue al carrito al agregar un producto

Confirmar navegación

- Si queremos “cancelar” la navegación desde un componente, o solicitar confirmación al usuario, podemos usar **withRouter** con ese componente, y después establecer la función a la que queremos que nos llame (ej. en **componentDidMount**):

```
this.props.router.setRouteLeaveHook(ruta, hookFunction)
```

- Si el componente es un RouteComponent, dispone de la ruta actual en **this.props.route**
- Si devolvemos un texto en la función *hookFunction*, el navegador lo mostrará como confirmación. “¿Seguro que quiere “?”
- Si devolvemos **false** se cancela directamente
- Útil para formularios a medio completar

Confirmar navegación

```
componentDidMount() {  
  //router Leave hook (route, function)  
  this.props.router.setRouteLeaveHook(this.props.route, this.handleRouteLeave);  
}  
  
handleRouteLeave(nextLocation) {  
  //solo si es el usuario ha escrito algo  
  const { details } = this.props;  
  if(details.firstName || details.lastName || details.email || details.address){  
    return "¿Seguro que quiere abandonar el proceso?";  
  }  
}
```

Suponemos que este componente está envuelto con **withRouter** y es el componente "destino" de alguna ruta

Navegar desde código

- El decorador **withRouter** nos sirve para navegar desde un componente, y para cancelar la navegación si hace falta
- Pero, ¿y si queremos navegar desde un action creator?
- ¿O si queremos consultar la ruta actual antes de despachar una acción en un thunk?

Integrar React Router con Redux

- Mantener sincronizado estado global con "history"
- Poder alterar el histórico mediante acciones
- Nos harán falta un reducer y un middleware, respectivamente

react-router-redux

- Una mini librería para mantener el state de Redux coordinando con la ruta
- **npm install -S react-router-redux**
- Nos ofrece:
 - Un reducer para combinar con los nuestros
 - Un middleware para integrar acciones con History
 - Action creators para despachar a este middleware

react-router-redux

- Añadimos el middleware en nuestro **configureStore**
- Y el reducer lo incluimos en **combineReducers** junto con nuestros módulos

react-router-redux

// configureStore.js

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


import catalog from './modules/catalog';

import cart from './modules/cart';

import order from './modules/order';

import thunkMiddleware from 'redux-thunk';

 **import** { hashHistory, browserHistory } from 'react-router';

 **import** { routerMiddleware, routerReducer } from 'react-router-redux';

import logger from './middlewares/logger';

 **const** router = routerMiddleware(browserHistory);

export default function configureStore() {

const appReducer = combineReducers({

 catalog,


 cart,

 order,

 routing: routerReducer

 });

//con devtools

 **return** createStore(appReducer, compose(

 applyMiddleware(thunkMiddleware, router),

 window.devToolsExtension ? window.devToolsExtension() : f => f)

);

}

react-router-redux

- Necesitamos "envolver" el history de modo que los cambios generen acciones para este nuevo reducer
- Este nuevo history será el que pasemos al Router
- La librería lo hace por nosotros con **syncHistoryWithStore**

react-router-redux

//app.js

```
import React from 'react';  
import { render } from 'react-dom';  
import { Provider } from 'react-redux';  
import configureStore from './configureStore';  
→ import { browserHistory, hashHistory } from 'react-router';  
import { syncHistoryWithStore } from 'react-router-redux';  
import Ecommerce from './components/ecommerce';
```

```
→ const store = configureStore();  
const history = syncHistoryWithStore(browserHistory, store);  
render(  
→
```

```
  <Provider store={ store }>  
    <Ecommerce history={ history } />  
  </Provider>  
  ,  
  document.getElementById('app')  
) ;
```

react-router-redux

- Ahora ya podemos usar los action creators de la librería
- **push(location)**
- **replace(location)**
- **go(number)**
- **goBack()**
- **goForward()**

react-router-redux

```
import { push } from 'react-router-redux';  
//...  
  
handleAddToCart(id) {  
  //Añadir producto al carrito  
  this.props.dispatch(addToCart(id));  
  //y luego navegar a la página del carrito  
  this.props.dispatch(push('/cart'));  
}
```

Ejercicio - terminar rutas

- Tendremos que sustituir nuestra navegación "manual" por las distintas alternativas que hemos visto (Link, withRouter, action creators)
- Añadir al carrito
- Volver al catálogo
- Ir al Checkout, etc...

React-router avanzado

- Hemos visto el uso básico de React Router
- Pero ofrece mucho más:
 - Decidir dinámicamente el componente de una ruta
 - Devolver varios componentes en un ruta, para incluir en distintas "regiones" de un layout padre
 - Cancelar la navegación para evitar que el usuario pierda datos (formulario)
 - Render en el servidor
 - <https://github.com/reactjs/react-router>

Acceso a APIs REST

- Ni React ni Redux nos ofrecen una solución lista para usar
- Porque depende mucho de cada proyecto
- Caso básico: cargar /enviar JSON desde el servidor

Acceso a APIs REST

- Existe un nuevo estándar para reemplazar XHR: **fetch**
- Implementado ya en Chrome y Firefox actuales, podemos usar polyfill para navegadores antiguos
- **npm install -S isomorphic-fetch**
- Esta librería nos sirve tanto para browser como para node.js

fetch - GET

- fetch nos devuelve Promises (se resuelven con la respuesta, se rechazan con error)

```
import fetch from 'isomorphic-fetch';

export function get(url) {
  return fetch(url).then(response => {
    if(response.ok) return response.json();
    throw {
      status: response.status,
      text: response.statusText
    }
  });
}
```

fetch - POST

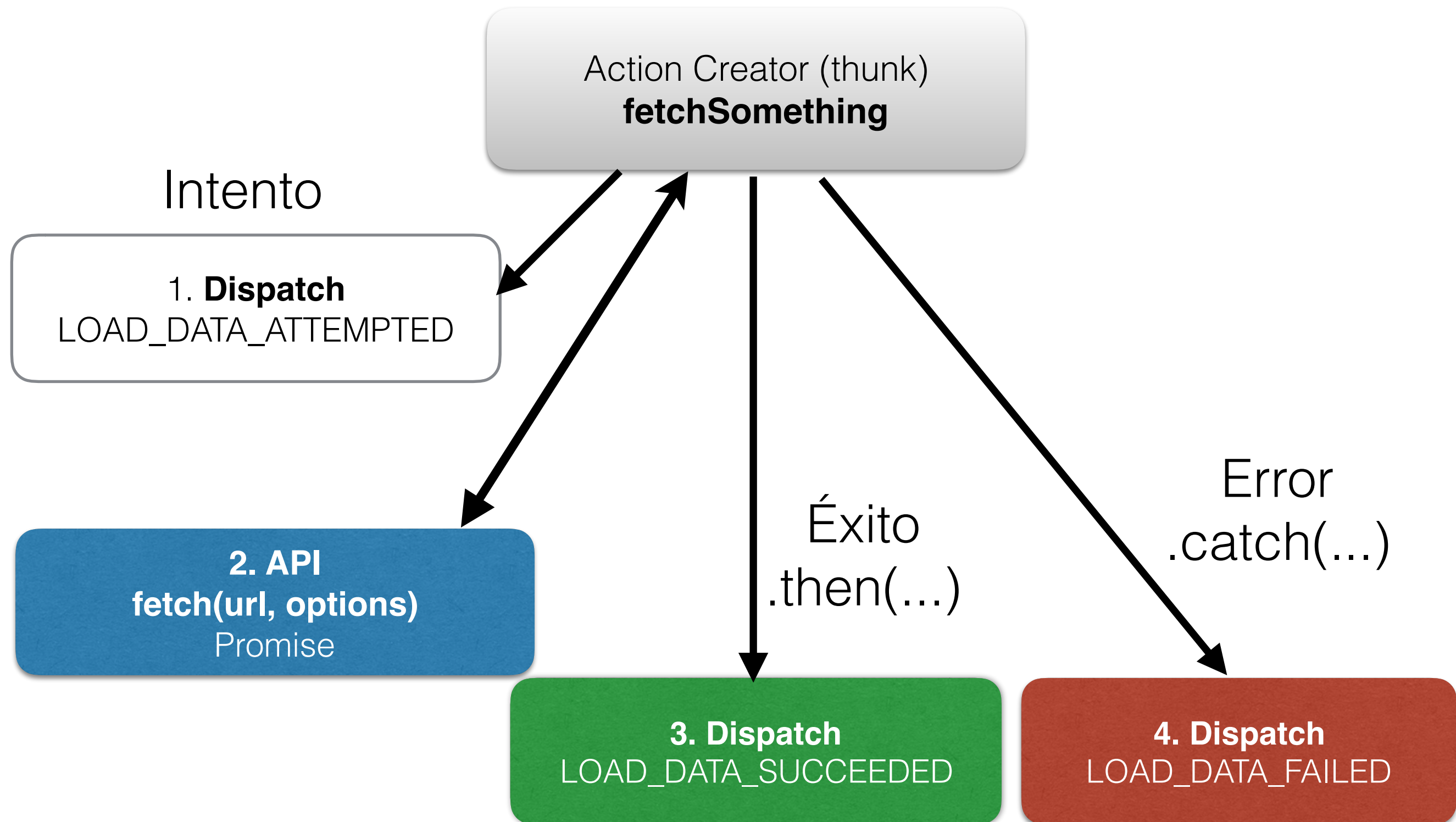
- Para enviar datos tenemos que ampliar la configuración un poco más

```
fetch('/users', {  
  method: 'POST',  
  headers: {  
    'Accept': 'application/json',  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({  
    name: 'Hubot',  
    login: 'hubot',  
  })  
})
```

Acciones asíncronas con Redux

- Tenemos el mecanismo para hacer las llamadas (fetch), ¿pero cómo integramos ésto con Redux?
- Lo importante en una llamada asíncrona es... petición, éxito, fracaso.
- Lo modelaremos así, con 3 action types (mínimo 2)
- Usaremos los **thunks**

Acciones asíncronas con Redux



Acciones asíncronas con Redux Action Creator

```
import { get } from '../lib/api';

export function fetchSomething() {
  return (dispatch, getState) => {
    dispatch({
      type: LOAD_DATA_ATTEMPTED
    });

    get('/api/resource')
      .then(data => {
        dispatch({
          type: LOAD_DATA_SUCCEEDED,
          payload: data
        });
      })
      .catch(err => dispatch({
        type: LOAD_DATA_FAILED,
        error: err
      }));
  }
}
```

Acciones asíncronas con Redux Reducer

```
//... import constantes, etc
const initialState = {
  data: [],
  isFetching: false,
  error: {}
}

export default function myReducer(state=initialState, action){
  switch(action.type){
    case LOAD_DATA_ATTEMPT:
      return {
        ...state,
        isFetching: true
      }
    case LOAD_DATA_SUCCEED:
      return {
        data: [ ...action.payload ],
        isFetching: false,
        error: {}
      }
    case LOAD_DATA_ERROR:
      console.error('Error fetching data', action.error.status, action.error.text);
      return {
        data: [],
        isFetching: false,
        error: action.error
      };
    default:
      return state;
  }
}
```

Ejercicio - cargar catálogo de forma asíncrona

- Vamos a implementar esta idea con nuestro catálogo
- Si hay error, pintaremos un mensaje de error (podemos probarlo poniendo una URL errónea)
- No tenemos un back, pero podemos cargar archivos JSON desde el servidor (en **/dist/api**) para probar
- Listo en el esqueleto (ejercicios/tema5/dist/api/products.json)

Acceso a APIs REST - Normalización

- Las APIs reales incluyen normalmente entidades anidadas

```
[{
  id: 1,
  title: 'Some Article',
  author: {
    id: 1,
    name: 'Dan'
  }
}, {
  id: 2,
  title: 'Other Article',
  author: {
    id: 1,
    name: 'Dan'
  }
}]
```

Acceso a APIs REST - Normalización

- ¿Qué problemas nos puede dar?
- Datos duplicados
- Acceso al autor "a través de" artículo
- Cambios -> datos no consistentes

Acceso a APIs REST - Normalización

- Sería mucho mejor así en el Store

```
entities: {
  articles: {
    1: {
      id: 1,
      title: 'Some Article',
      author: 1
    },
    2: {
      id: 2,
      title: 'Other Article',
      author: 1
    }
  },
  users: {
    1: {
      id: 1,
      name: 'Dan'
    }
  }
}
```

Acceso a APIs REST - Normalización

- <https://github.com/paularmstrong/normalizr>
- npm: **normalizr**
- Nos permite definir **esquemas** para las respuestas de una API

Acceso a APIs REST - Normalización

- <https://github.com/paularmstrong/normalizr>
- npm: **normalizr**
- Nos permite definir **esquemas** para las respuestas de una API

Acceso a APIs REST - normalizr

```
import { normalize, Schema, arrayOf } from 'normalizr';

// creamos esquemas para las dos entidades
const article = new Schema('articles');
const user = new Schema('users');
// definimos la relación entre artículo y usuario
article.define({
  author: user,
  contributors: arrayOf(user)
});

//..
//al recibir JSON, llamamos a normalize con el esquema apropiado

var response = normalize(data, article);
```

Acceso a APIs REST - normalizr

```
{  
  id: 1,  
  title: 'Some Article',  
  author: {  
    id: 7,  
    name: 'Dan'  
  },  
  contributors: [{  
    id: 10,  
    name: 'Abe'  
  },  
  {  
    id: 15,  
    name: 'Fred'  
  }]  
}
```

Acceso a APIs REST - normalizr

```
{
  result: 1,
  entities: {
    articles: {
      1: {
        author: 7,
        contributors: [10, 15],
        title: 'Some article'
      },
    },
    users: {
      7: { id: 7, name: 'Dan' },
      10: { ... },
      15: { ... }
    }
  }
}
```

Acceso a APIs REST - normalizr



```
{
  result: 1,
  entities: {
    articles: {
      1: {
        author: 7,
        contributors: [10, 15],
        title: 'Some article'
      },
    },
    users: {
      7: { id: 7, name: 'Dan' },
      10: { ... },
      15: { ... }
    }
  }
}
```

Reducer con datos normalizados

- Podemos usar **un único reducer para guardar todas las entidades de la aplicación**
- Solo con establecer un patrón al despachar acciones con los resultados normalizados

Reducer con datos normalizados

```
{  
  type: 'LOAD_ARTICLES_SUCCEEDED',  
  payload: {  
    result,  
    entities  
  }  
}
```

Reducer con datos normalizados

- En este reducer "entities" atenderemos cualquier acción que incluya **entities**
- Es nuestro super diccionario global de entidades

Reducer con datos normalizados

```
function entities(state={ articles: {}, users: {} }, action){  
  if(action.payload && action.payload.entities){  
    return {  
      ...state,  
      users: { ...state.users, ...action.payload.entities.users },  
      articles: { ...state.articles, ...action.payload.entities.articles }  
    }  
  }  
  
  return state;  
}
```

Reducer con datos normalizados

```
function articles(state=[], action) {  
  if(action.type === LOAD_ARTICLES_SUCCEEDED) {  
    return action.payload.result  
  }  
  
  return state;  
}
```

Reducer con datos normalizados

```
function mapStateToProps (state) {  
  const { articles } = state.entities;  
  const articleIds = state.articles;  
  
  return {  
    articles: articleIds.map(id => articles[id])  
  }  
}
```

Resumen

npm install

configu

middleware

connec

push

spatchToProps

combineRed

withRouter

createSelector

browserHistory

fetch

import { ... }

syncHistoryWithStore

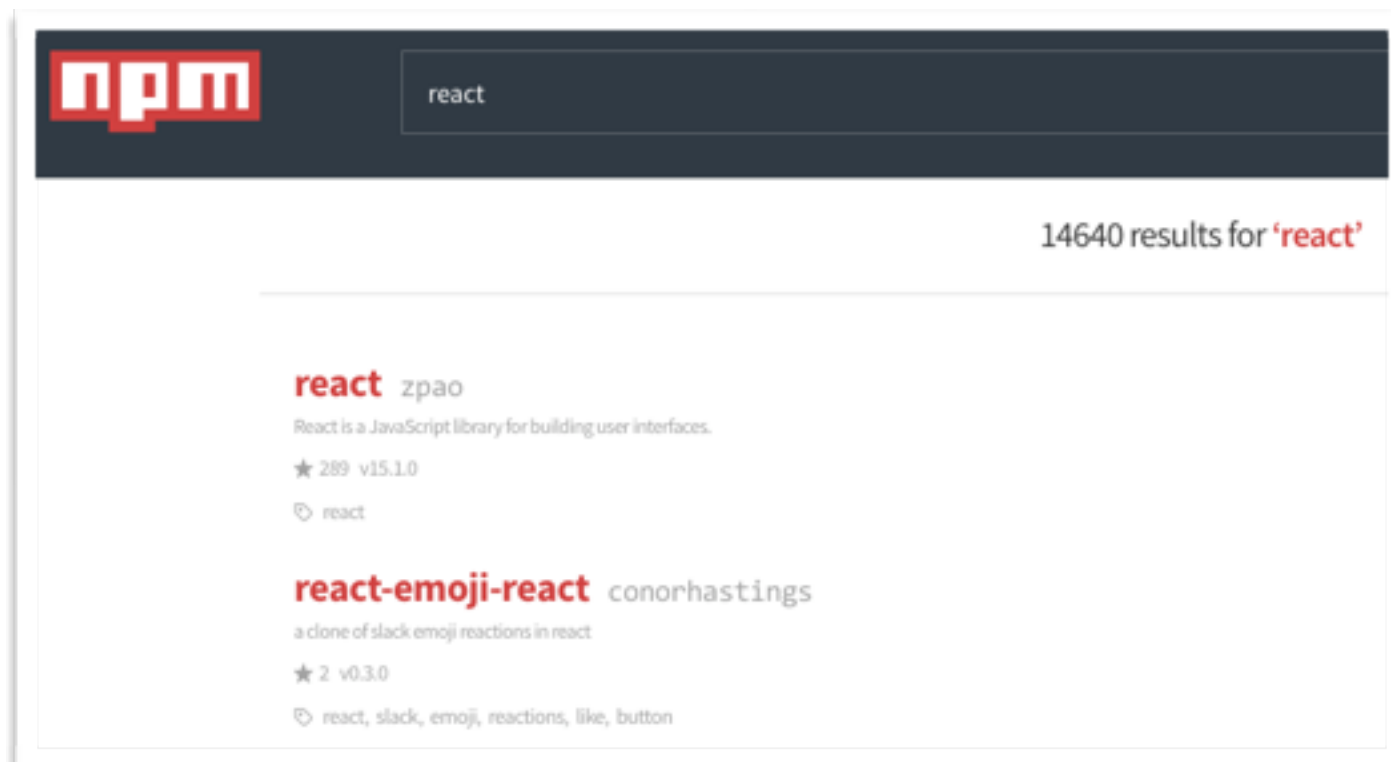


Resumen

- React (y Redux) es un ecosistema más que un framework
- Compuesto por micro-librerías
- Para que usemos lo que necesitamos y nada más



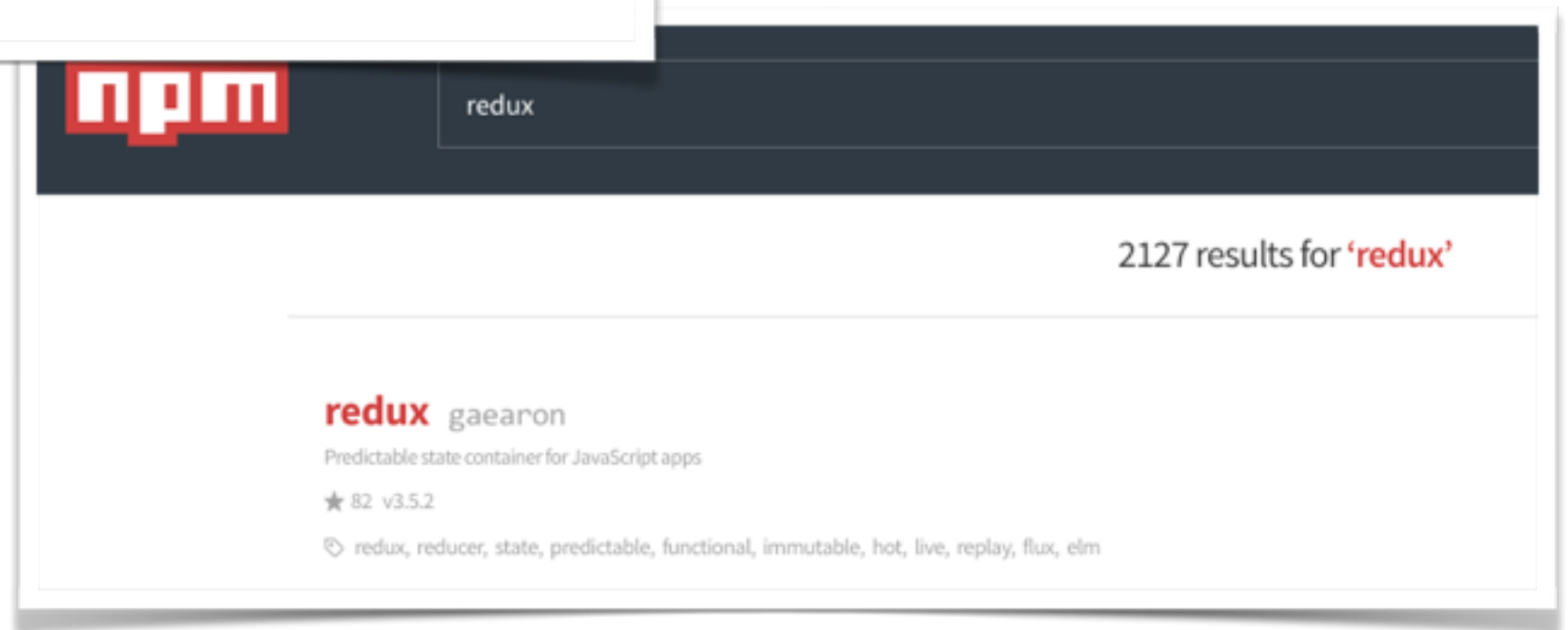
Resumen



The screenshot shows the NPM search interface for the term 'react'. The NPM logo is in the top left, and the search bar contains 'react'. The results section shows 14640 results. The top two results are 'react' by 'zpao' and 'react-emoji-react' by 'conorhastings'.

react zpao
React is a JavaScript library for building user interfaces.
★ 289 v15.1.0
🔗 react

react-emoji-react conorhastings
a clone of slack emoji reactions in react
★ 2 v0.3.0
🔗 react, slack, emoji, reactions, like, button



The screenshot shows the NPM search interface for the term 'redux'. The NPM logo is in the top left, and the search bar contains 'redux'. The results section shows 2127 results. The top result is 'redux' by 'gaearon'.

redux gaearon
Predictable state container for JavaScript apps
★ 82 v3.5.2
🔗 redux, reducer, state, predictable, functional, immutable, hot, live, replay, flux, elm