Redux

Introducción

- Redux es una **librería**
- Para gestionar el **estado de la aplicación**
- Agrupa **todo el estado** en un árbol central
- Regula las **transformaciones** que se pueden aplicar

Introducción

- ¿Qué es el **estado** de una aplicación?
  - El **conjunto de datos** necesarios para representar la la aplicación **en un momento dado**
  - Que mutan en el tiempo
    - Por **interacción** del usuario
    - Por la **lógica** de negocio

Tres principios de Redux

- **Todo el estado** de la aplicación se guarda en **un solo objeto** central.
- El árbol de estado **no se puede modificar** directamente.
- Hay que escribir **funciones** que expresen las **transformaciones** que se pueden aplicar al estado

Tres principios de Redux

- **Árbol de estado central**
  - En Redux se llama **store**
  - Contiene **todo el estado** de la aplicación
  - Organizado jerárquicamente
  - **Única fuente de verdad**
  - Los componentes **pueden leer** su contenido

Tres principios de Redux

- **El estado NO se puede modificar**
  - Podemos expresar la **intención de modificarlo**
  - Utilizando **acciones predeterminadas**
  - Cada acción representa **una transformación**
  - Las **acciones** tienen una estructura específica

Tres principios de Redux

- **Reducers aplican los cambios al estado**
  - **Reducers** son funciones que expresan el efecto de aplicar una acción al estado
  - Por tanto, expresan **las transformaciones** permitidas
  - Son **funciones puras**

Hola, Mundo

redradix

Requisitos

- Necesitamos:
  - Infraestructura para compilar React
  - `redux`
  - `react-redux`

Requisitos

```
$ npm install -S redux react-redux
```

redradix

```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

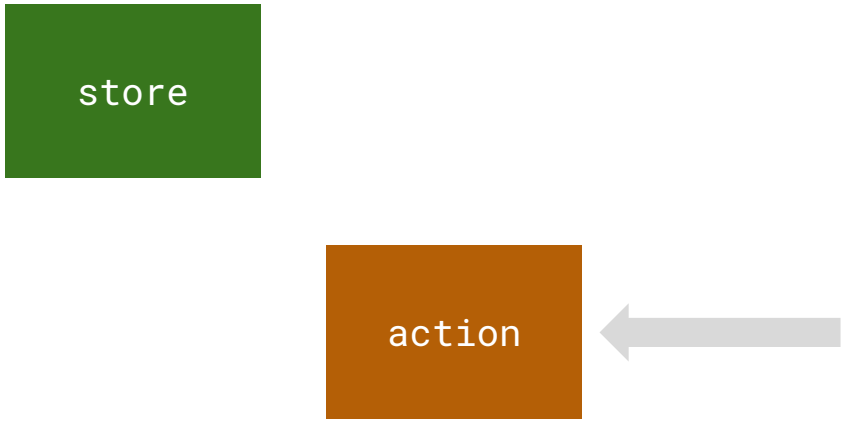
```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

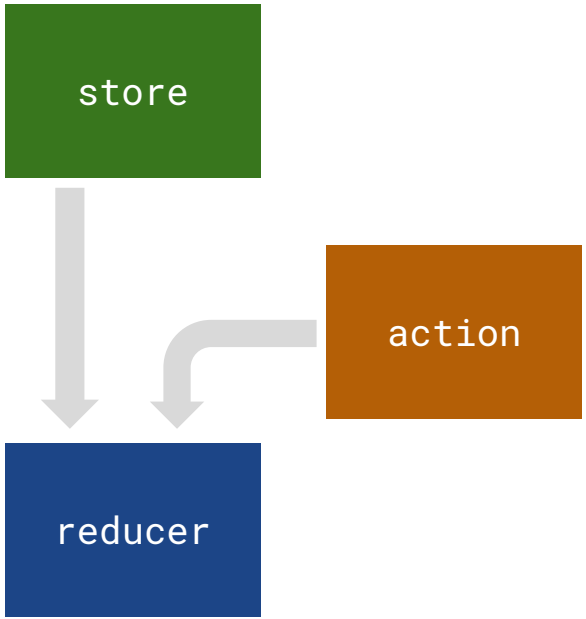
```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

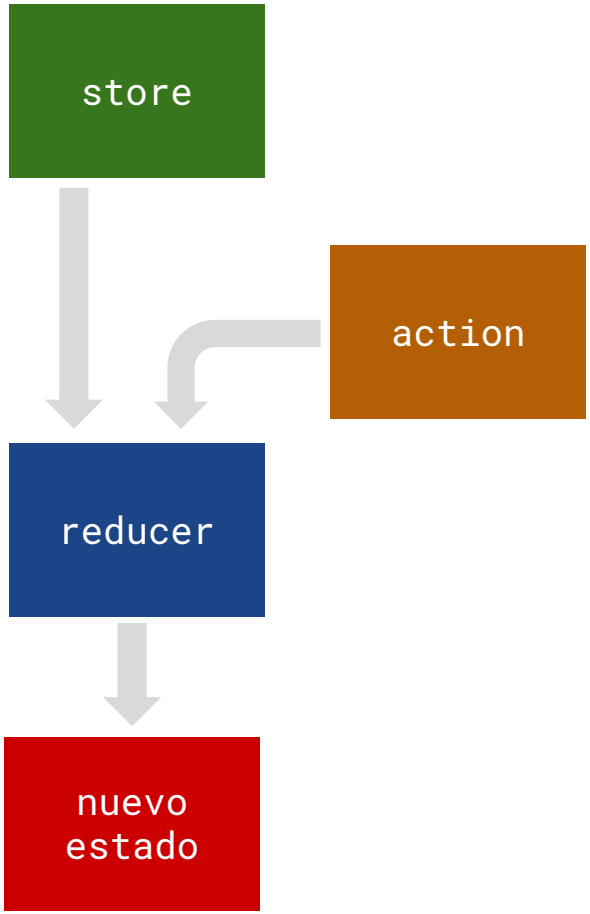
```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

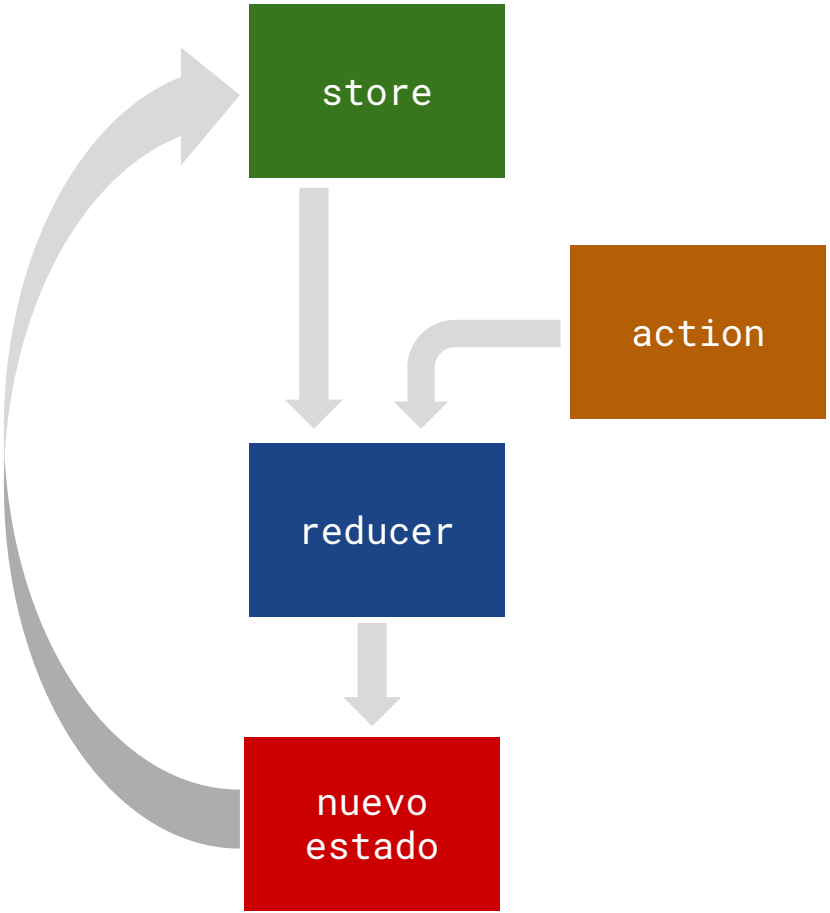
redradix







redradix



redradix

```
import { createStore } from 'redux'

// reducer

function counter(state, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

// store

const store = createStore(counter, 0)

store.subscribe(() => {
  console.log(`-> nuevo estado: ${store.getState()}`)
})

// action

store.dispatch({ type: 'INCREMENT' })
```

Ejercicio

- Partiendo de /001
  - **Modifica** el código para que el contador se pueda **decrementar**

Ejercicio

- Partiendo de /001
  - **Modifica** el código para poder manejar **DOS** contadores a la vez
  - Ambos contadores guardan su valor en el **store**

Acciones

- Las acciones son **objetos**
- Que tienen una propiedad **`type`**
- Pero pueden tener otras propiedades!
  - **`payload`**
  - **`text`**
  - **`...`**

```
store.dispatch({
  type: 'ADD',
  payload: 5
})
```

Ejercicio

- Partiendo de /001
  - **Modifica** el código para que se puedan sumar o restar diferentes cantidades a cada uno de los dos contadores

Ejercicio

- Partiendo de cero...
  - Crea un store que sea una **cola**
  - Escribe el reducer para que se puedan **añadir** y **sacar** elementos de la cola
  - Escribe unas cuantas acciones de prueba

Combinar Reducers

Combinar Reducers

- Escribir un **reducer** por módulo
- Cada uno maneja **una rama** del estado
- Expresando sus transformaciones **independientemente**
- Todos los reducers **reciben todas las acciones**

Combinar Reducers

`redux/002/index.js`

```
function counter(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

function stack(state = [], action) {
  switch(action.type) {
    case 'PUSH':
      return [...state, action.payload]
    default:
      return state
  }
}
```

```
function counter(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

function stack(state = [], action) {
  switch(action.type) {
    case 'PUSH':
      return [...state, action.payload]
    default:
      return state
  }
}
```

```
const store = createStore(combineReducers({
  stack: stack,
  counter: counter
}))
```

```
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })

store.dispatch({ type: 'PUSH', payload: 'Hola' })
```

Combinar Reducers

¿Qué pasa si quitamos el **default** de los **switch** de los reducers?

Combinar Reducers

¿Qué pasa si definimos el store así?

```
const store = createStore(combineReducers({
  counter: stack,
  stack: counter
}))
```

Combinar Reducers

¿Qué pasa si definimos el store de esta forma?

```
const store = createStore(combineReducers({
  stack1: stack,
  stack2: stack,
  counter1: counter,
  counter2: counter
}))
```

Combinar Reducers

¿Qué pasa pasamos un **segundo parámetro**?

```
const store = createStore(combineReducers({
  stack: stack,
  counter: counter
}), { stack: [0], counter: 10 })
```

Ejercicio

- Partiendo del ejercicio anterior...
  - añade un reducer **books** en el que se puedan
    - **crear** libros
    - **eliminar** libros
    - **modificar** libros

Conectando con React

Conectando con React

`redux/003/index.js`

```
const Counter = (props) => <h1> {props.value} </h1>
```

```
function counter(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

const store = createStore(combineReducers({ counter }))
```

```
store.subscribe(() => {
  const state = store.getState()
  ReactDOM.render(
    <Counter value={state.counter} />,
    document.getElementById('app')
  )
})

// action
window.onload = () => {
  store.dispatch({ type: 'INCREMENT' })
  store.dispatch({ type: 'INCREMENT' })
}
```

```
store.subscribe(() => {
  const state = store.getState()
  ReactDOM.render(
    <Counter value={state.counter} />,
    document.getElementById('app')
  )
})

// action
window.onload = () => {
  store.dispatch({ type: 'INCREMENT' })
  store.dispatch({ type: 'INCREMENT' })
}
```

```
store.subscribe(() => {
  const state = store.getState()
  ReactDOM.render(
    <Counter value={state.counter} />,
    document.getElementById('app')
  )
})

// action
window.onload = () => {
  store.dispatch({ type: 'INCREMENT' })
  store.dispatch({ type: 'INCREMENT' })
}
```

Ejercicio

- Modifica el ejemplo anterior...
  - añade dos botones: **incrementar** y **decrementar**
  - que disparen dos acciones, **INCREMENT** y **DECREMENT**, que cambien el valor del contador

react-redux

react-redux

- Dos ideas fundamentales:
  - **Provider**
  - **connect**

react-redux

- **Provider**
  - nos permite asociar un árbol de componentes con un **store**

```
const store = createStore(combineReducers({ counter }))

ReactDOM.render(
    <Provider store={store}>
      <App />
    </Provider>,
  document.getElementById('app')
)
```

react-redux

- **connect**
  - nos permite asociar un componente a valores del **store**
  - el componente se **re-rendea automáticamente cuando esos valores cambian**

Conectando con React

`redux/004/index.js`

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore, combineReducers } from 'redux'
import { Provider, connect } from 'react-redux'
```

```
function counter(state = 0, action) {
  switch(action.type) {
    case 'INCREMENT':
      return state + 1
    default:
      return state
  }
}

const store = createStore(combineReducers({ counter }))
```

redradix

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button
        onClick={() => this.props.dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })

const ConnectedCounter = connect(mapStateToProps)(Counter)
```

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button
        onClick={() => this.props.dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })

const ConnectedCounter = connect(mapStateToProps)(Counter)
```

redradix

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button
        onClick={() => this.props.dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })

const ConnectedCounter = connect(mapStateToProps)(Counter)
```

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button
        onClick={() => this.props.dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })

const ConnectedCounter = connect(mapStateToProps)(Counter)
```

```
window.onload = () => {
  ReactDOM.render(
    <Provider store={store}><ConnectedCounter/></Provider>,
    document.getElementById('app')
  )
}
```

Combinar Reducers

¿Qué pasa si definimos el store así?

```
const store = createStore(combineReducers({
  contador: counter
}))
```

Combinar Reducers

¿Qué pasa si definimos **mapStateToProps** así?

```
const mapStateToProps = state => ({
 counter: state.counterValue
})
```

Ejercicio

- Modifica el ejemplo anterior...
  - Extrae el componente **Display** y el componente **Button**
  - Añade dos **Button**: uno para incrementar y otro para decrementar
  - Añade un **Button** “reset” que ponga el contador a cero

redradix

Dumb vs. Smart

- Los componentes **conectados** se llaman **containers** o **smart componentes**
- Los componentes **no conectados** se llaman **dumb components** o **componentes presentacionales**
- Presentacionales > Contenedores

react-redux

- **connect**
  - recibe un **segundo parámetro**
  - **mapDispatchToProps**
    - una función que recibe **dispatch** como parámetro
    - devuelve un objeto con props que se pasarán al componente

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button
        onClick={() => this.props.dispatch({ type: 'INCREMENT' })}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })
const ConnectedCounter = connect(mapStateToProps)(Counter)
```

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button onClick={this.props.increment}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })
const mapDispatchToProps = dispatch => ({
  increment: () => dispatch({ type: 'INCREMENT' })
})
const ConnectedCounter = connect(mapStateToProps, mapDispatchToProps)(Counter)
```

```
class Counter extends React.Component {
  render() {
    return <div>
      <h1>{this.props.value}</h1>
      <button onClick={this.props.increment}>
        Increment
      </button>
    </div>
  }
}

const mapStateToProps = state => ({ value: state.counter })
const mapDispatchToProps = dispatch => ({
  increment: () => dispatch({ type: 'INCREMENT' })
})
const ConnectedCounter = connect(mapStateToProps, mapDispatchToProps)(Counter)
```

Ejercicio

- Partiendo de ejercicio anterior...
  - Modifica el código para tener **múltiples** contadores en la página
  - Cada uno con sus botones de incremento, decremento y reset

Ejercicio

- Partiendo de cero...
  - Implementa un **reloj** con Redux
  - Componentes:
    - Clock
    - Segment

Ejercicio

- Partiendo de cero...
  - Implementa el **cronómetro** del tema anterior con Redux
  - Piensa en **qué reducers** tienes que implementar

módulos

Módulos

- La lógica de la ap se suele dividir en módulos
- Cada módulo tiene...
  - un **reducer**
  - un fichero con **actionTypes**
  - un fichero con **actionCreators**
  - (opcional) un fichero con **selectores**

Módulos

- **ActionTypes**
  - Simplemente los **type** de las acciones que acepta el reducer exportados como **constantes**
  - Para evitar erratas

Módulos

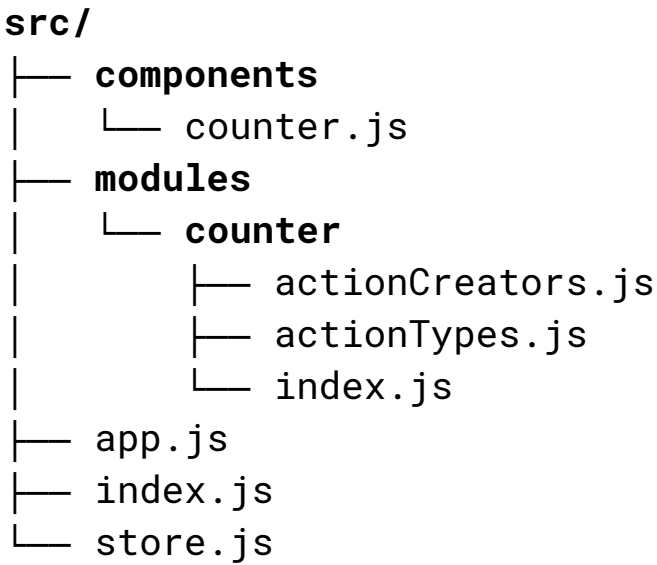
- **ActionCreators**
  - Funciones que construyen y configuran **acciones**
  - Para facilitar la vida al programador
  - Son un “*interfaz*” (informal) del módulo

Módulos

- **Selectores**
  - Funciones que **extraen** valores del estado
  - Útiles en reducers que manejan estructuras de datos complejas
  - Se utilizan desde **mapStateToProps**
  - Suelen recibir el **state** como primer parámetro

Módulos

`redux/005/index.js`


src/
components
counter.js
modules
counter
actionCreators.js
actionTypes.js
index.js
app.js
index.js
store.js

Ejercicio

- Aplica esta estructura de ficheros al **cronómetro**
  - Cada componente en su fichero
  - Divide el store en **módulos**
  - Utiliza **actionCreators**

Ejercicio

- Modifica el **cronómetro** para que guarde **laps**
  - Añade un botón **LAP**
    - activo cuando el cronómetro está en marcha
    - añade el tiempo actual a una lista de tiempos
    - en cada fila de la lista, un botón **DELETE**
  - Añade un botón **RESET**
    - borra la lista de tiempos

Módulos

- Se pueden hacer módulos de uso general
- Por ejemplo:
  - un **módulo** que guarde **flags** de la interfaz
  - se puede reutilizar para todos los componentes
  - añadiendo propiedades a la acción

```
import { createStore } from 'redux'

const SET_FLAG = 'SET_FLAG'

const reducer = (state = {}, action) => {
  const { name, payload } = action
  switch(action.type) {
    case SET_FLAG:
      return Object.assign({}, state, { [name]: payload })
    default:
      return state
  }
}

const store = createStore(reducer)

store.dispatch({
  type: 'SET_FLAG',
  name: 'chrono-active',
  payload: true
})
```

Ejercicio

- Modifica el **cronómetro**...
  - Completa el **reducer** del ejemplo anterior y escribe su módulo
  - Utilízalo para simplificar el resto del cronómetro

Ejercicio

- Modifica el **cronómetro**...
  - Añade la acción **DELETE_FLAG**
  - Modifica el contenedor del cronómetro para que **limpie sus flags** del módulo cuando sea desmontado

formularios

redradix

Ejercicio

- Partiendo de cero...
  - Crea una aplicación con un único componente **TextInput**
  - Gestionado por **redux**

Ejercicio

- Modifica el ejercicio anterior...
  - Para que la aplicación gestione 5 **TextInput**
  - Pista: **se puede automatizar?**

Ejercicio

- Modifica **TextInput** para que reciba como prop
  - el **nombre** del campo
  - una **función de validación**
    - devuelve mensaje de error o null
  - muestre el **mensaje de error** cuando no es válido

Ejercicio

- Partiendo de **redux/006**
  - Implementa la funcionalidad del **To Do**
  - Empieza:
    - Identificando componentes
    - Identificando módulos
    - Programa **toda la lógica** en redux antes de programar los componentes

asincronía

Asincronía

- Todo lo que hemos hecho hasta ahora es síncrono
- Cómo podemos gestionar un proceso asíncrono con la infraestructura que tenemos?
  - Por ejemplo, una llamada AJAX

Ejercicio

- Partiendo de **redux/007**
  - Escribe un módulo que haga una petición a `api.openwathermap.org` para consultar el tiempo que hace en **Madrid**
  - Escribe un componente que utilice el módulo y muestre el resultado

Middleware

- Funciones que modifican el comportamiento de **dispatch**
- Tenemos que **“instalarlas”** al configurar el store
- Hay una buena colección de middlewares interesantes
  - logging
  - control de errores
  - ....

Asincronía

- **redux-thunk**
  - Nos permite construir *action creators* asíncronos
  - Despachando una **función** en vez de un objeto
  - Esa función se invoca inmediatamente y recibe **dispatch** como parámetro

Asincronía

`redux/008/index.js`

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore, combineReducers, applyMiddleware } from 'redux'
import { Provider, connect } from 'react-redux'
import thunk from 'redux-thunk'
```

```
const store = createStore(
  combineReducers({ counter }),
  applyMiddleware(thunk)
)
```

```
store.dispatch({ type: 'INCREMENT' })

store.dispatch((dispatch) => {
  console.log('dispatch asincrono!')
  setTimeout(() => dispatch({ type: 'INCREMENT' }), 100)
})
```

Ejercicio

- Partiendo de **redux/007**
  - **`npm install -S redux-thunk`**
  - Escribe un módulo que haga una petición a `https://query.yahooapis.com` para consultar el tiempo que hace en **Madrid**
  - **https://developer.yahoo.com/weather/**
  - Escribe un componente que utilice el módulo y muestre el resultado

Ejercicio

- Modifica el ejercicio anterior
  - Añade un **TextInput** y un botón para que el usuario pueda seleccionar la ciudad que quiere consultar
  - Muestra un **Spinner** mientras carga el resultado
  - Muestra un **Error** si la ciudad no existe

Ejercicio

- Modifica el ejercicio anterior
  - Modifica la aplicación para que muestre una **lista de ciudades**
  - El usuario puede **añadir** y **eliminar ciudades de la lista**
  - Cada ciudad muestra **su meteorología**