

TRYBE

Modulo II – Front-end

Bloco 16 – Redux

Definição

Biblioteca de estados criada com princípios da arquitetura [flux](#) criada por Facebook.

Criadores: *Dan Abramov e Andrew Clark*

[Documentação oficial Redux](#).

1) Introdução ao Redux (com Js puro)

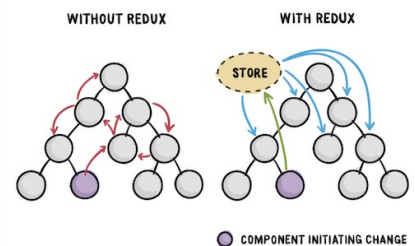
Propósito

Feito para desenvolvimento Web, particularmente UI.

Roda em diversos ambientes: servidor, cliente, nativo.

Biblioteca para gerenciamento de estados de uma aplicação em Javascript. Resolve problema de *prop threading (drilling)* do React, melhorando comunicação entre componentes.

Aumenta organização e legibilidade do código.



Composição

Store: grande objeto, container que armazena de forma centralizada os estados. Imutável.

Actions: ações disparadas da aplicação pelo store, criadas via *action creators*.

Única forma de mudar estados do Store.

Reducers: especificam como acontece essa atualização.



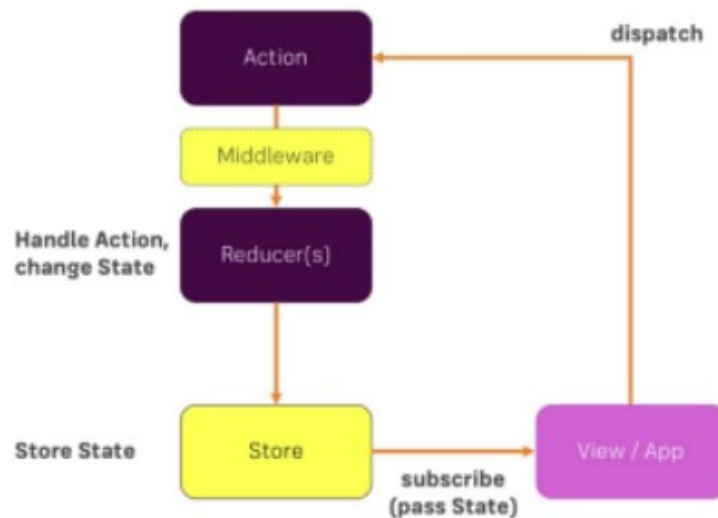
Princípios

- Store é fonte da verdade e de todos estados (*"This means if you had a React app with ten components, and each component had its own local state, the entire state of your app would be defined by a single state object housed in the Redux store"*);
- Estados são apenas leitura (*"In Redux, all state updates are triggered by dispatching actions"*);
- Alterações feitas por funções puras: reducers. Pega estado anterior + ação e retorna novo estado.

Uso

Pode ser usado sozinho diretamente no Js (Lib *Redux Starter Kit*) mas o normal é usar junto com frameworks como Angular, Ember, Vue. O mais comum: com React. Lib oficial *React-Redux*.

Primeiros passos: etapas e sintaxes do ciclo Redux



//View: UI variando dependendo do State.

Instalar o Redux

→ Importando no html

→ OU usando npm

npm install redux

```
<script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
```

Criar o Store

const store = Redux.createStore(paramReducer)

Store é único e tem as responsabilidades seguintes:

- Allow access to state via `getState()`.
- Allow state to be updated via `dispatch(action)`.
- Holds the whole application state.
- Registers listeners using `subscribe(listener)`.
- Unregisters listeners via the function returned by `subscribe(listener)`.

Get State do Store

const currentState = store.getState();

Enviar uma Action

→ action é um *objeto* que contém o *type*, e as vezes, um *payload* (valor).

const action = { type: 'LOGIN' }

→ Criar action via Action Creator

```
export function addTodo({ task }) {  
  return {  
    type: 'ADD_TODO',  
    payload: {  
      task,  
      completed: false  
    },  
  },  
}
```

- event listener
- dispatchar uma Action para o Reducer com *store.dispatch(paramObjectContainingActionInfo)*.

Tratar a Action dentro do Reducer

- Reducer é uma function pura com 2 parâmetros: estado atual e action do store.dispatch().

```
function myReducer(previousState, action) => {
  // use the action type and payload to create a new state based on
  // the previous state.
  return newState;
}
```

Fechar fazendo Subscription

- Function executada cada vez que uma Action for disparada.

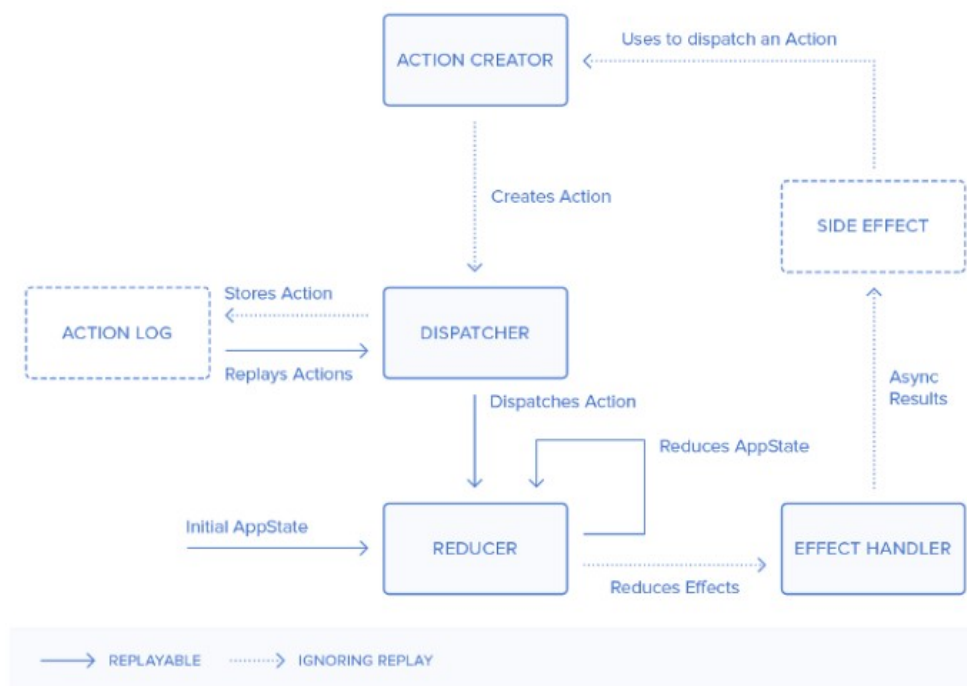
store.subscribe(render);

store.subscribe(function) OU store.subscribe() => writefunctionhere);

Resumindo: data flow em Redux

4 etapas principais:

- 1/ Evento do app provoca uma chamada para o store.dispatch(actionCreator(payload))
- 2/ O store chama o reducer com estado atual e ação
- 3/ O reducer combina o output de reducers dentro de um novo estado
- 4/ O store salva essa nova árvore completa de estado retornada pelo reducer principal, que vira o nextState ou seja estado atualizado do aplicativo.



Redux/Flux Flow

Porque e quando utilizar o Redux

Motivos de uso (5)

- Previsível graças ao seu fluxo unidirecional;
- Ferramentas de desenvolvimento: hot reloading, time travel, record e replay;
- Fácil para testar: testes de actions, reducers, middleware, components. Sem mock por conta das functions serem puras;
- Velocidade de desenvolvimento;
- Existe [curso de graça do criador](#).

Momento de uso

Usar quando efeitos colaterais e aplicativo viram grandes demais.

Pode prever e começar com Redux, ou pode adaptar aplicação integrando aos poucos Redux.

Não usar se conseguir resolver com Context API do React.

`combineReducers(reducers)`

A **rootReducer** é a que pode ser passada como parâmetro da `createStore` e pode ser uma combinação de functions Reducer separadas menores.

rootReducer = Redux.combineReducers({potato: potatoReducer, tomato: tomatoReducer})

// This would produce the following state object

```
{
  potato: {
    // ... potatoes, and other state managed by the potatoReducer ...
  },
  tomato: {
    // ... tomatoes, and other state managed by the tomatoReducer, maybe some nice sauce? ...
  }
}
```

Preservar imutabilidade do estado (no reducer)

A/ Quando state é array

→ Diferenciar métodos de *mudança* que modificam o array e os de *clonagem* que retornam novo: [link](#).

Mutação:

`.push()`, `.splice()`, `.sort()`

Clonagem:

`.concat()`, `.slice()`, `.filter()`,
spread operator `[...array]`

ARRAY CHEATSHEET

```
['a', 'b'].concat(['c']) //['a','b','c']
['a', 'b'].join('~') // 'a~b'
['a','b','c'].slice(1) //['b', 'c']
['a','b','b'].indexOf('b') // 1
['a','b','b'].lastIndexOf('b') //2

['a','b','c'].forEach(x => console.log(x))

[1,2,3].every(x => x < 10) //true
[1,2,3].some(x => x < 2) //true
[1,2,3].filter(x => x < 2) //[1]

[1, 2, 3].map(x => x * 2) //[2, 4, 6]
[1, 2, 3].reduce((x,y) => x * y) //6
[2, 15,3].sort()//[ 15, 2, 3 ]
[1, 2, 3].reverse()//[ 3, 2, 1 ]
[1, 2, 3].length //3

const arr = [1, 2, 3]
const x=arr.shift()//arr=[ 2, 3 ],x=1
const x=arr.unshift(9)//arr=[ 9,1,2,3 ],x=4
const x=arr.pop()//arr=[ 1, 2 ],x=3
const x=arr.push(5)//arr=[1,2,3,5],x=4

const arr=['a','b','c','d'];const mod = arr.splice(1,2,'z');//arr=['a','z','d'],mod=['b','c']
```

B/ Quando state é objeto

Object.assign

const newObject = Object.assign({}, obj1, obj2);

Dicas diversas

Usar **Switch** para gerir diferentes Actions

```
const authReducer = (state =
defaultState, action) => {
  // change code below this line
  switch (action.type) {
    case "LOGIN":
      return {
        authenticated: true
      };

    case "LOGOUT":
      return {
        authenticated: false
      };

    default:
      return defaultState;
  }
}
```

—

Ver app Redux de contador:

```
const INCREMENT = "INCREMENT"; // define a constant for increment action types
const DECREMENT = "DECREMENT"; // define a constant for decrement action types
```

```
// define the counter reducer which will increment or decrement the state based on the action it receives
```

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case INCREMENT:
      return state + 1;
    case DECREMENT:
      return state - 1;
    default:
      return state;
  }
};
```

```
// define an action creator for incrementing
```

```
const incAction = () => {
  return {
    type: INCREMENT
  };
};
```

```
// define an action creator for decrementing
```

```
const decAction = () => {
  return {
    type: DECREMENT
  };
};
```

```
// define the Redux store here, passing in your reducers
```

```
const store = Redux.createStore(counterReducer);
```

```
//Também era possível adicionar uma propriedade value nos objetos para somar action.value no Reducer.
```

Resumo do dia 16-1:

State

É onde vamos armazenar todos os dados da aplicação e é representado por um objeto *JavaScript*.
O *State* é armazenado no **Store** do **Redux**.

Action

É um objeto *JavaScript* que representa alguma mudança/alteração que precisa acontecer no **State**.

Reducer

É uma função *JavaScript* que recebe o estado atual (*current state*) e a ação corrente (*current action*) e retorna um novo estado (*new state*). É responsabilidade dessa função decidir o que acontecerá com o estado dada uma ação (*action*).

Dispatch

É uma função que envia uma ação (*action*) para processamento.

2) React com Redux - Parte I

Configurar Redux com React

→ Criar React app

```
npx create-react-app my-app  
cd my-app
```

→ Instalar dependências

```
npm install react-redux / npm i react-redux / npm install --save redux react-redux
```

→ Estrutura de arquivos

A/ Criar a estrutura pura do Redux: *store*, *reducers* e *actions*.

* criar o store no `src/store/index.js`

```
import { createStore } from 'redux'; // depois adicionar mais import e colocar reducer em param  
  
const store = createStore(); // boa prática mesmo que com apenas um Reducer:  
const rootReducer = combineReducers({ arqReducer });  
  
export default store;
```

* criar reducer no `src/reducer/index.js`

* guardar actions num arquivo `src/actions/index.js`

B/ Conectar com React: *provider*, *connect*, *dispatch*, *ToProps* e *mapStateToProps*.

* No App.js: importar Provider e store do Redux e integrar ambos no render

```
1 import React from 'react';
2 // o provider é o meio pelo qual disponibilizamos o Store
3 import { Provider } from 'react-redux';
4 import store from './store';
5
6 class App extends React.Component {
7   render() {
8     return (
9       <div>
10         <Provider store={store}>
11           // componentes aqui
12         </Provider>
13       </div>
14     );
15   }
16 }
17
18 export default App;
```

* Conectar componentes

- com o App React, com o importar de sempre
- neles mesmo, import e export do Redux no início e fim do arquivo:

```
import React from 'react';
import { connect } from 'react-redux';
import { addAssignment } from './actions';
```

```
const mapDispatchToProps = dispatch => ({
  add: e => dispatch(addAssignment(e))});

export default connect(null, mapDispatchToProps)(InputsList);
```

- componente ainda pode ter estado mudado via evento dentro.

mapDispatchToProps

Dispara (faz o dispatch de) uma ação para o reducer. Criado via function (arrow/convencional).

Termos que acessar funcionalidades do Redux como props de componente
= usar mapDispatchToProps que mapeia os dispatchs para os props.

Syntax

```
const mapDispatchToProps = dispatch => ({
  MyPropAsKey: e => dispatch(importedAction(e))});
```

Segundo parâmetro opcional: ownProps

```
const mapDispatchToProps = (dispatch, ownProps) => {
  const dispatchFunction = ownProps.cake
    ? () => dispatch(buyCake())
    : () => dispatch(buyIceCream())

  return {
    buyItem: dispatchFunction
  }
}
```


As duas escrituras seguintes são equivalentes:

```
// const mapDispatchToProps = (dispatch) => ({  
  // changeSignal: (payload) => dispatch(changeSignal(payload))});  
const mapDispatchToProps = { changeSignal };
```

Behind the scenes, React Redux is using Redux's `store.dispatch()` to conduct these dispatches with `mapDispatchToProps()`.

connect

Da acesso ao store do Redux.

Syntax

```
export default connect(myNativeReduxMethod)(someComponent);
```

Método nativo Redux sendo escrito nesta ordem, dependendo do caso:

`(mapStatetoProps, null) / (null, mapDispatchToProps) / (mapStatetoProps, mapDispatchToProps)`

mapStateToProps

Mapeia as entidades armazenadas nos estados para uma props.

Criado via function (arrow/convencional).

Syntax

```
const mapStateToProps = state => ({  
  MyPropAsKey: state.myReducer});
```

Segundo parâmetro opcional: `ownProps`
(no exemplo, `item` e `cake` são props do comp →)

```
const mapStateToProps = (state, ownProps) => {  
  const itemState = ownProps.cake  
    ? state.cake.numOfCakes  
    : state.iceCream.numOfIceCreams  
  
  return {  
    item: itemState  
  }  
}
```

As duas escrituras seguintes são equivalentes, com ou sem arrow, com ou sem return:

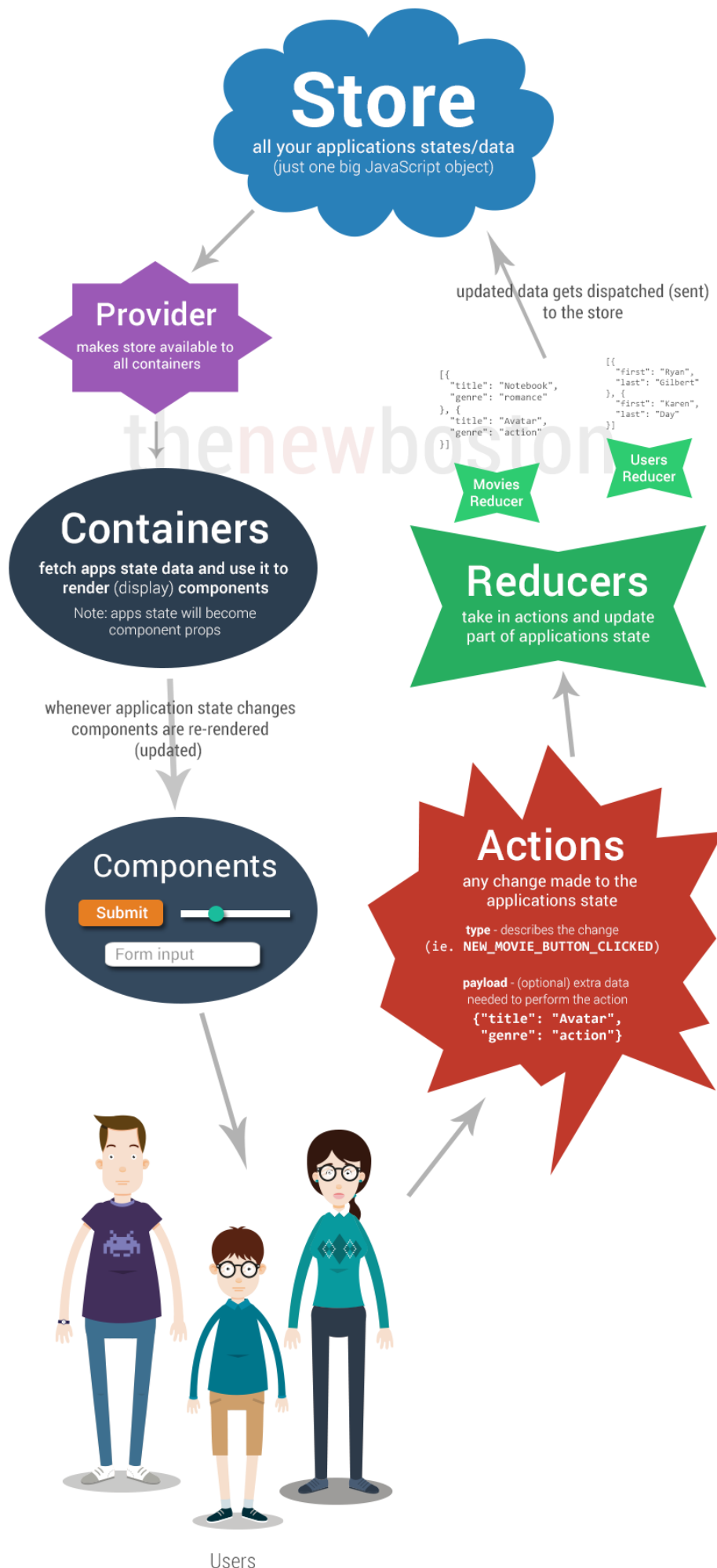
```
// const mapStateToProps = (state) => ({ signalColor: state.trafficReducer.signal.color })  
function mapStateToProps(state) {  
  return { signalColor: state.trafficReducer.signal.color }  
}
```

Behind the scenes, React Redux is using Redux's `store.subscribe()` method to implement `mapStateToProps()`.

Resumo final: fluxo de dados no Redux

1. Um **Store** é criado para armazenar todos o estado da aplicação;
2. O **Store** é disponibilizado através do **Provider** para todos os componentes da aplicação;
3. Os componentes usam o **connect** para conectarem-se ao **Store**;
4. As pessoas que utilizam a aplicação interagem com ela e disparam eventos;
5. Estes eventos têm o nome de **Actions** e são enviadas ao **Store** através de um **dispatch**;
6. Os **Reducers** recebem essas **Actions** e alteram o estado da aplicação (criando um novo estado) e salvando no **Store**;
7. Os componentes conectados ao **Store** "ouvem" estas mudanças e atualizam a **View** (visualização).

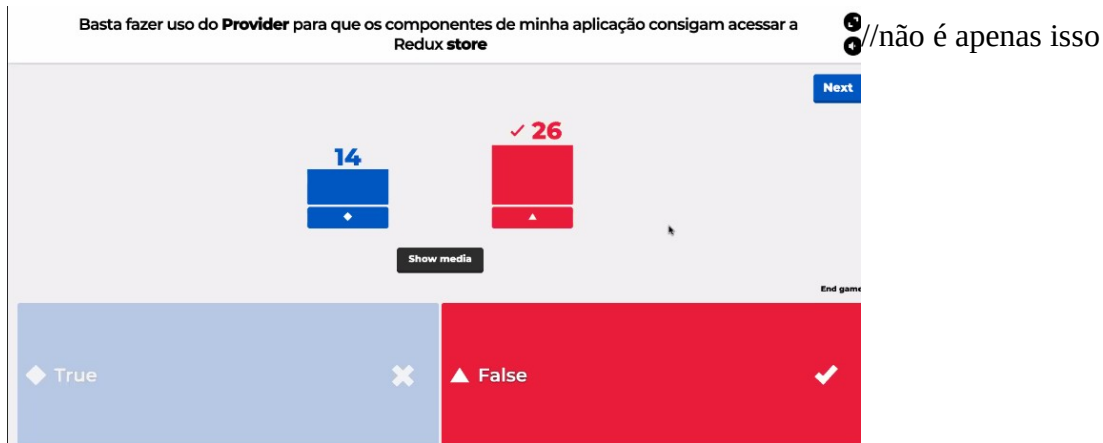
Redux Explained



Dicas diversas

Kahoot

Basta fazer uso do **Provider** para que os componentes de minha aplicação consigam acessar a **Redux store** // não é apenas isso

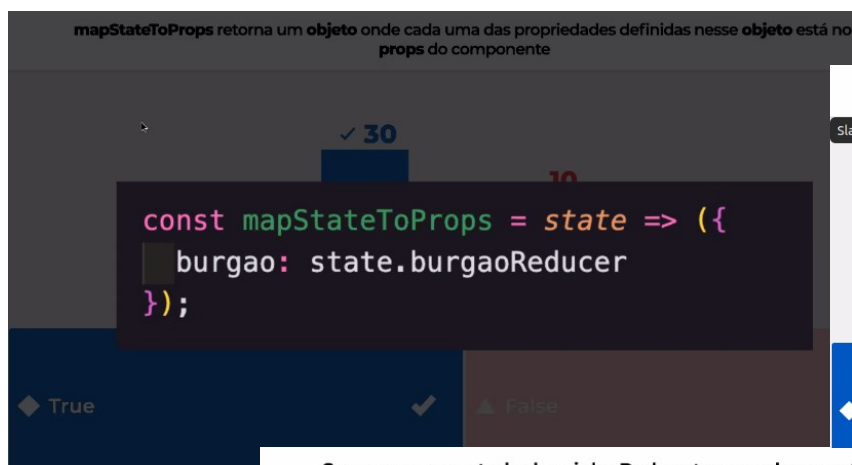


Next

End game

`mapStateToProps` retorna um **objeto** onde cada uma das propriedades definidas nesse **objeto** está no **props** do componente

```
const mapStateToProps = state => ({
  burgao: state.burgaoReducer
});
```



Slack

Next

End game

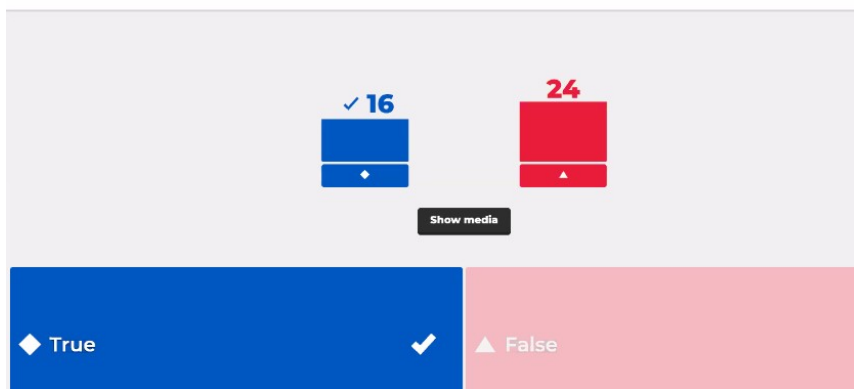
A função **connect** aceita uma **callback** que permite que você chame funções que enviam **actions** ao **store**



Next

End game

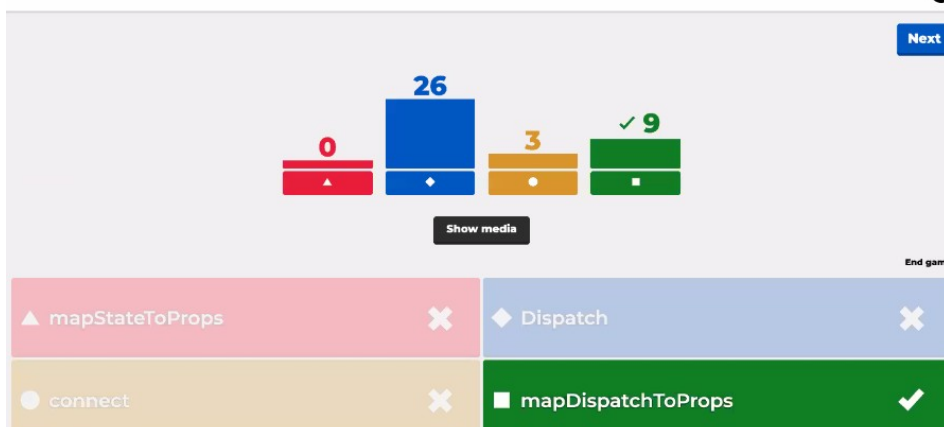
Sempre que o estado de minha Redux store mudar, **mapStateToProps** será chamado



Next

End game

Qual função do **react-redux** envia uma action para o store ?



Next

End game

// `mapDispatchToProps` para separar as actions de interesse para serem passadas como props para nosso componente

Extension Chrome Redux Devtools

→ Instalar extension no Browser

→ Instalar no aplicativo (ver [repositório fonte](#))

npm install --save redux-devtools-extension

```
import { createStore, applyMiddleware } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
```

```
const store = createStore(reducer, composeWithDevTools(
  applyMiddleware(...middleware),
  // other store enhancers if any
));
```

→ OU apenas com

```
const store = createStore(rootReducer, window.__REDUX_DEVTOOLS_EXTENSION__ &&
window.__REDUX_DEVTOOLS_EXTENSION__());
```

→ OU para não modificar dependências e deixar compatível com thunk, terceiro jeito:

```
import { createStore, compose, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from '../reducers';
```

```
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
```

```
const store = createStore(rootReducer, composeEnhancers(applyMiddleware(thunk)));
```

```
export default store;
```

Diferença entre **(presentational) component** e **container component**: container é o que fica conectado com Redux, envelopado no Provider no render do app principal.

Conectar os dois quando feitos aparte:

```
const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);
```

```
//Display it in the console
store.subscribe(() => console.log(store.getState()));

//DISPATCH
store.dispatch(increment());
store.dispatch(decrement());
```

useSelector to access the whole state within store

| useDispatch to dispatch action

```
import React from 'react';
import { useSelector } from 'react-redux';

function App() {
  const counter = useSelector(state => state.counter);
  return (
    <div className="App">
      <h1>Counter {counter}</h1>
    </div>
  );
}

export default App;
```

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment } from './actions';

function App() {
  const counter = useSelector(state => state.counter);
  const isLogged = useSelector(state => state.isLogged);
  const dispatch = useDispatch();
```

3) React com Redux - Parte I – Dia de prática

Spread operator

Bom quando lidarmos com imutabilidade, forma de criar algo novo a partir de algo existente.

Usar Redux?

“Vale sempre a reflexão sobre quando é interessante utilizar e quando pode ser melhor utilizar alguma outra tecnologia para controle de estado”.

4) React com Redux - Parte II: actions assíncronas

O pacote **Redux** suporta somente **fluxo síncrono de dados**, assim precisamos usar o **redux-thunk** que provê uma interface simples para dar suporte a action creators que geram actions assíncronas.

redux-thunk

Definição

“[Middleware](#) provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.”

Interceptador: captura as actions enviadas pelo store antes delas chegarem a um reducer.

Uso

Permite definir uma **action creator que retorna uma função** em vez de somente um objeto, útil para realizar uma operação assíncrona (como chamada a API) e enviar action integrando os dados obtidos.

Thunk

Função que encapsula uma operação para que ela seja feita posteriormente.

Parâmetros (*dispatch*, *getState*),
possibilidade de 3º via via [withExtraArgument](#) →

Olha para cada action que passa, se for uma function, chama essa function.

Resumindo:

“thunk nada mais é do que uma action que, quando despachada, faz uma requisição assíncrona e aguarda o resultado da requisição, podendo disparar uma ação em caso de sucesso (tratando as informações recebidas) ou disparando outra ação em caso de falha para buscar a informação.”

```
function logOutUser() {
  return function(dispatch, getState) {
    return axios.post('/logout').then(function() {
      // pretend we declared an action creator
      // called 'userLoggedOut', and now we can dispatch it
      dispatch(userLoggedOut());
    });
  };
}
```

```
const store = createStore(
  reducer,
  applyMiddleware(thunk.withExtraArgument(api)),
);

// later
function fetchUser(id) {
  return (dispatch, getState, api) => {
    // you can use api here
  };
}
```

```

export const REQUEST_MOVIES = 'REQUEST_MOVIES';
export const RECEIVE_MOVIES = 'RECEIVE_MOVIES';

const requestMovies = () => ({ // action creator que retorna um objeto, que você tem feito
  type: REQUEST_MOVIES});

const receiveMovies = (movies) => ({ // outro action creator que retorna um objeto, que você tem feito
  type: RECEIVE_MOVIES,
  movies});

export function fetchMovies() { // action creator que retorna uma função, possível por causa do thunk
  return (dispatch) => { // thunk declarado
    dispatch(requestMovies());
    return fetch('some API endpoint irrelevant for our purposes now')
      .then((response) => response.json())
      .then((movies) => dispatch(receiveMovies(movies)));
  };
}

// componente que você consumiria a action fetchMovies assíncrona, como uma outra qualquer
...
class MyConnectedAppToRedux extends Component {
  ...
  componentDidMount() {
    const { dispatch, fetchMovies } = this.props;
    dispatch(fetchMovies()); // enviando a action fetchMovies
  }
  ...
}
...

```

Instalar e habilitar

→ no terminal: *npm install redux-thunk / npm i redux-thunk / npm install --save redux-thunk*

→ no store:

```

// arquivo onde a redux store é criada
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducer from '/path/to/your/root/reducer';

...

const store = createStore(reducer, applyMiddleware(thunk));
...

```

Pequena Lib - code inteiro:

```

function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => next => action => {
    // This gets called for every action you dispatch.
    // If it's a function, call it.
    if (typeof action === 'function') {
      return action(dispatch, getState, extraArgument);
    }

    // Otherwise, just continue processing this action as usual
    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;

```

Dicas diversas

Kahoot



// usamos redux thunk justamente por isso

—

redux && render: jeito de condicionar segundo a veracidade ou existência de props na renderização

```
{isFetching && 'Loading...'}  
{!isFetching && isLocationPresent && `Current ISS Location:  
latitude: ${latitude}, longitude: ${longitude}`}  
{!isFetching && error}
```

5) Testes síncronos com React-Redux

Testando Redux

renderWithRedux

Parecida com renderWithRouter. Usada para simular o retorno do componente.
Function auxiliar para renderizar componente com redux antes de testar.

Syntax

```
const renderWithRedux = (  
  component,  
  { initialState, store = createStore(reducer, initialState) } = {}  
) => {  
  return {  
    ...render(<Provider store={store}>{component}</Provider>),  
    store,  
  }  
}
```

Parâmetros: (*componente*, *objetoDesconstruido*).

Retorno: renderiza componente envolvido pela store e também a própria store (objeto).

1/ Ajustes para ter ambiente funcional de testes

→ Criar pasta src/helpers e dentro um arquivo renderWithRender.js onde incluir a function

→ Imports

```
import React from 'react'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import { render, cleanup, fireEvent } from '@testing-library/react';
import App from './App';
import reducer from './reducers'
```

→ em caso de combineReducers, modificar renderWithRouter
`store = createStore(combineReducers({ clickReducer })`
(também importando todo o necessário)

→ mover o Provider do App.js para o index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';
import * as serviceWorker from './serviceWorker';

// Store provida pela nossa aplicação. Nos testes, precisamos prover um novo store

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);

serviceWorker.unregister();
```

//Assim ficamos livres para criar um novo store (mockar) que podemos controlar no ambiente de testes!

2/ Fazer testes

Arquivo App.test.js

Mesmo trabalho que com React Testing Library.

(Também tem que cuidar do histórico quando tiver navegação).

```
content.md JS App.test.js X
dux-tests > src > JS App.test.js > ...
1 import React from 'react';
2 import { fireEvent } from '@testing-library/react';
3 import { createMemoryHistory } from 'history';
4 import { Router } from 'react-router-dom';
5
6 import renderWithRedux from './helpers/renderWithRedux';
7
8 import App from './App';
9
10 describe('clients enrolment', () => {
11   const renderAppWithRouter = (initialEntries = ['/']) => (
12     <Router history={createMemoryHistory({ initialEntries })}>
13       <App />
14     </Router>
15   )
16 }
```


Possibilidade de **adicionar valor inicial** de uma prop no reducer.

```
const { queryByText } = renderWithRedux(<App />, { initialState: { myReducer: { myPropHereCounter: 5 } } });
```

```
test('renders initial page(HOME)', () => {
  const { getByText } = renderWithRedux(
    renderAppWithRouter(),
    { initialState: { registerReducer: [], loginReducer: {} } }
  )

  const textOfPage = getByText('Bem-vindo ao sistema de cadastramento!');
  expect(textOfPage).toBeInTheDocument();
})
})
```

//O App vai sair do zero, aqui estamos forçando que já começa com esses estados
//Depois segue com testes clássicos

```
fireEvent.change(inputEmail, { target: { value: 'usuario001' } });
expect(inputEmail.value).toBe('usuario001');
```

//Outro exemplo forçado

5) Projeto Star Wars

Dicas diversas

→ Error “data undefined” pode ser problema da API
Tentar abrir o link da API e ver código HTTP.

Planet List

GET /api/planets/

HTTP 429 TOO MANY REQUESTS

Retry-After: 63729

Content-Type: application/json

Vary: Accept

Allow: GET, HEAD, OPTIONS

```
{
  "detail": "Request was throttled. Expected available in 63729 seconds."
}
```

Codes: 429 too many requests / 400 error / 200 OK.

Solucionar: reiniciar seu modem internet ou usar do celular.

—
→ Escolha: estados entre React e Redux

Lembrar de usar ainda estados locais de componente React quando pertinente e menos trabalhoso (exemplo req.3).

—
Select: para dar titulo sem ser uma primeira option seleccionável

```
{/* <option hidden disabled selected value>By Column</option> */}
```

`/* <option hidden disabled selected value>By Comparison</option> */`

Input: com placeholder

placeholder="Insert value"

—

CSS Fonte: `@font-face` para integrar nova fonte baixada.

```
@font-face {
  font-family: StarWarsFont;
  src: url(../images/Starjedi.ttf);
}

.title {
  font-family: StarWarsFont;
}
```

6) Projeto Trivia

Dicas diversas

→ *JSON.stringify* para converter dado em string no contexto de passar ele para servidor:

https://www.w3schools.com/js/js_json_stringify.asp

→ *JSON.parse* para modificar de volta para objeto:

https://www.w3schools.com/js/js_json_parse.asp .

Possibilidade de substituir
`mapStateToProps` e
`mapDispatchToProps` por:

```
export default function GameButton({ isAvailable, click }) {
  const dispatch = useDispatch();
  const settings = useSelector(state => state.settingsReducer);
  console.log(settings);
}
```

CHEATSHEET Redux

```
react-redux connect
import { connect } from 'react-redux'

YourComponent = connect(
  mapStateToProps,
  mapDispatchToProps
)(YourComponent)

export default YourComponent
```


```
Store
import {
  createStore,
  combineReducers
} from 'redux'
import todos from './todosReducer'
import counter from './counterReducer'

const rootReducer = combineReducers({
  todos,
  counter
})

const store = createStore(rootReducer)

export default store
```

React Redux

 @PrasoonPratham

```
reducers
const initialState = {
  todos: []
}

function todosReducer(state = initialState, action) {
  switch (action.type) {
    case UPDATE_TODO:
      const newState = deepClone(state)
      const todo = newState.todos.find(
        todo => todo.id === action.id
      )
      todo.text = action.text
      return newState
  }
}

function deepClone(obj) {
  return JSON.parse(JSON.stringify(obj))
}
```

```
action creators
const addTodo = (text) => ({
  type: ADD_TODO,
  text
})
const removeTodo = (id) => ({
  type: REMOVE_TODO,
  id
})
const updateTodo = (id, text) => ({
  type: UPDATE_TODO,
  id,
  text
})
```

```
react-redux provider
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

const store = createStore(todoApp)

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

```
Action Types
const ADD_TODO = 'ADD_TODO'
const REMOVE_TODO = 'REMOVE_TODO'
const UPDATE_TODO = 'UPDATE_TODO'
```