

{LET'S CONTEXT EVERYTHING}



# Aula #4 - Let's Context Everything

- ❑ **High Order Components** – HOC
- ❑ **Fragments**
- ❑ **Context API**
- ❑ **PropTypes**
- ❑ **Componentes controlados e não controlados**
- ❑ **Redux**

{HOCs}

# o que são HOCs?

é uma técnica avançada para reutilizar a lógica de um componente. Um padrão que surgiu no modo de composição do React. É uma função que recebe um componente e retorna um NOVO componente.



```
const NewComponent = higherOrderComponent(WrappedComponent)
```



```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" é uma fonte de dados global
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscreve-se às mudanças
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // Limpa o listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Atualiza o state do componente sempre que a fonte de dados muda
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}
```



```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

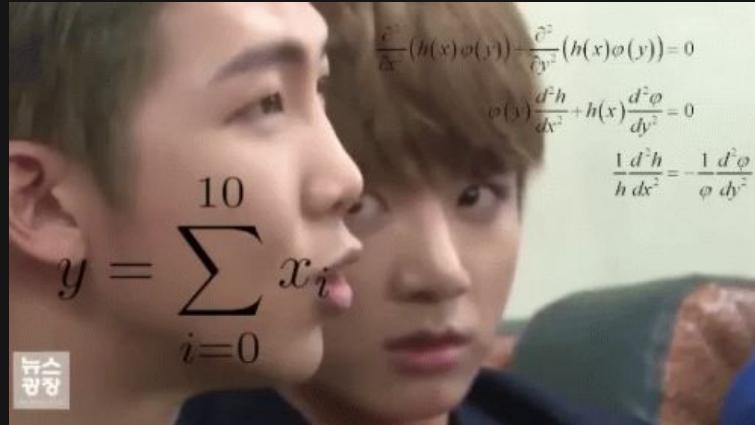
  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

- ❑ A parte lógica dos componentes são similares
- ❑ Mesmo padrão de subscriver-se a DataSource e chamar setState
- ❑ Esse mesmo padrão, em uma aplicação grande, pode se repetir várias vezes

- ❑ A parte lógica dos componentes são similares
- ❑ Mesmo padrão de subscrever-se a `DataSource` e chamar `setState`
- ❑ Esse mesmo padrão, em uma aplicação grande, pode se repetir várias vezes

Podemos isolar a parte lógica com HOCs



vamos criar uma função que criar componentes e executa toda a lógica com DataSource



```
const CommentListWithSubscription = withSubscription(  
  CommentList,  
  (DataSource) => DataSource.getComments( )  
)  
  
const BlogPostWithSubscription = withSubscription(  
  BlogPost,  
  (DataSource, props) =>  
  DataSource.getBlogPost(props.id)
```

vamos criar uma função que criar componentes e executa toda a lógica com DataSource



Componente encapsulado

```
const CommentListWithSubscription = withSubscription(  
  CommentList,  
  (DataSource) => DataSource.getComments( )  
)
```

```
const BlogPostWithSubscription = withSubscription(  
  BlogPost,  
  (DataSource, props) =>  
  DataSource.getBlogPost(props.id)
```

# vamos criar uma função que criar componentes e executa toda a lógica com DataSource



Componente encapsulado

```
const CommentListWithSubscription = withSubscription(  
  CommentList,  
  (DataSource) => DataSource.getComments( )  
)
```

Acessa DataSource

```
const BlogPostWithSubscription = withSubscription(  
  BlogPost,  
  (DataSource, props) =>  
  DataSource.getBlogPost(props.id)
```



```
// A função recebe um componente...
function withSubscription(WrappedComponent, selectData) {
  // ...e retorna outro componente...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... que lida com a subscrição...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... e renderiza o componente encapsulado com os dados novos!
      // Note que nós passamos diretamente qualquer prop adicional
      return <WrappedComponent data={this.state.data} {...this.props}
    }
  };
}
```



```
// A função recebe um componente...
function withSubscription(WrappedComponent, selectData) {
  // ...e retorna outro componente...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(dataSource, props)
      };
    }

    componentDidMount() {
      // ... que lida com a subscrição...
      dataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      dataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(dataSource, this.props)
      });
    }

    render() {
      // ... e renderiza o componente encapsulado com os dados novos!
      // Note que nós passamos diretamente qualquer prop adicional
      return <WrappedComponent data={this.state.data} {...this.props}
    />;
  };
}
```

- ❑ **não mudamos o componente encapsulado**
- ❑ **o componente encapsulado recebe todas as props necessárias e até mesmo não relacionadas ao HOC**
- ❑ **Container components pattern**
- ❑ **e é isso!**

# onde encontrar HOCs?

- ❑ **Redux**
- ❑ **React Router**
- ❑ **Formik**
- ❑ **Apollo**
- ❑ **Relay**

{fragments}

# o que são Fragments?

Um padrão comum no React para retornar múltiplos elementos. Fragments permitem agrupar uma lista de filhos sem adicionar nós extras ao DOM.

# por que precisamos disso?

Failed to compile

```
./src/teste.js
Line 7:13:  Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>?

 5 |       return (
 6 |         <Component></Component>
> 7 |         <h1>testeee</h1>
   |         ^
 8 |       )
 9 |
10 | }
```

This error occurred during the build time and cannot be dismissed.

# como usar?

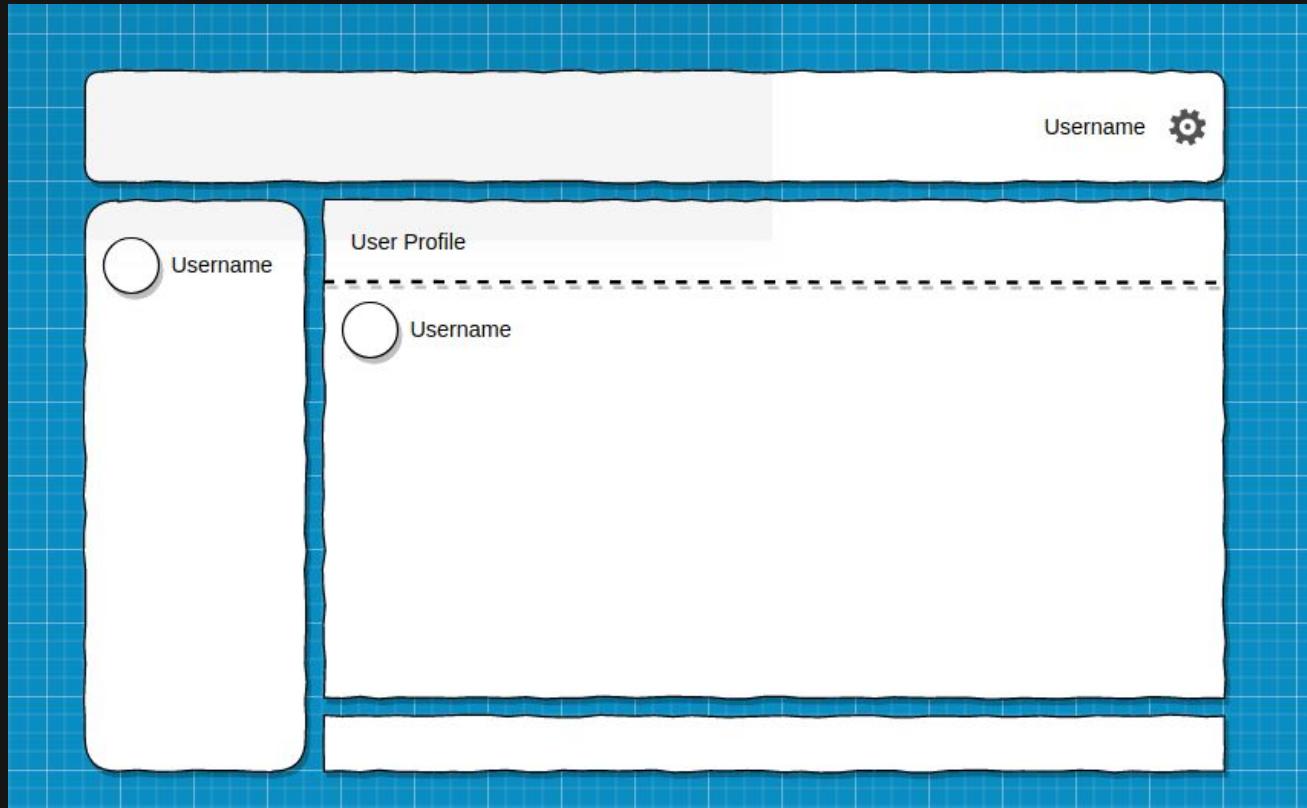
```
render() {  
  return (  
    <Fragment>  
      <ChildA />  
      <ChildB />  
      <ChildC />  
    <Fragment>  
  )  
}
```

```
render() {  
  return (  
    <>  
      <Component/>  
      <Component2/>  
    </>  
  )  
}
```

{context API}

# o que é Context API?

Em React, props são passadas de cima para baixo (de pai para filho). Em certos cenários, principalmente em uma aplicação muito grande, props utilizadas por muitos componentes podem dar um pouco de dor de cabeça para controlar. A context API fornece a forma de compartilhar dados como esses, entre todos componentes da mesma árvore de componentes, sem precisar passar explicitamente props entre cada nível.



- body (div#app)
  - app
    - header
      - **settings** (vai usar a propriedade **user**)
    - sidebar
      - sidebar-header
        - **user-avatar** (vai usar a propriedade **user**)
    - content
      - content-header
      - content-body
        - **user-avatar** (vai usar a propriedade **user**)
    - footer

# como usar?

- **primeiro criamos um contexto com `createContext(string|object)`, recebe como arg o valor default do contexto.**



```
const MyContext = React.createContext(defaultValue)
```

# como usar?

- ❑ todo objeto contexto vem com um componente Provider que permite com que os componentes que “assinam” esse contexto possam acessar suas mudanças



```
<MyContext.Provider value={/* some value */}>
```

# como usar?

- ❑ é o componente que consome o contexto fornecido pelo provider.



```
<MyContext.Consumer>
  {value => /* renderiza algo baseado no valor do context */}
</MyContext.Consumer>
```



## Exercício

{PropTypes}

# **o que é?**

**Auxilia na documentação e validação das propriedades que um componente recebe.**

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

```
MyComponent.propTypes = {  
  optionalArray: PropTypes.array,  
  optionalBool: PropTypes.bool,  
  optionalFunc: PropTypes.func,  
  optionalNumber: PropTypes.number,  
  optionalObject: PropTypes.object,  
  optionalString: PropTypes.string,  
  optionalSymbol: PropTypes.symbol,  
  optionalElement: PropTypes.element,  
  optionalElementType: PropTypes.elementType  
}
```

```
● ● ●
```

```
class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.defaultProps = {
  name: 'Stranger'
};

ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

{componentes controlados  
e não controlados}

# **o que são?**

**Componentes controlados possuem seus valores manipulado pelo React e componentes não controlados são manipulados apenas pela DOM.**

# componentes não controlados

```
class Form extends Component {  
  render() {  
    return (  
      <div>  
        <input type="text" />  
      </div>  
    );  
  }  
}
```

# componentes não controlados

```
● ● ●

class Form extends Component {
  handleSubmitClick = () => {
    const name = this._name.value;
    alert('welcome ' + name)
  }

  render() {
    return (
      <div>
        <input type="text" ref={input => this._name = input} />
        <button onClick={this.handleSubmitClick}>Sign up</button>
      </div>
    );
  }
}
```

# componentes controlados



```
<input value={someValue} onChange={handleChange} />
```

```
class Form extends Component {
  constructor() {
    super();
    this.state = {
      name: '',
    };
  }

  handleNameChange = (event) => {
    this.setState({ name: event.target.value });
  };

  render() {
    return (
      <div>
        <input
          type="text"
          value={this.state.name}
          onChange={this.handleNameChange}
        />
      </div>
    );
  }
}
```

# qual eu uso?

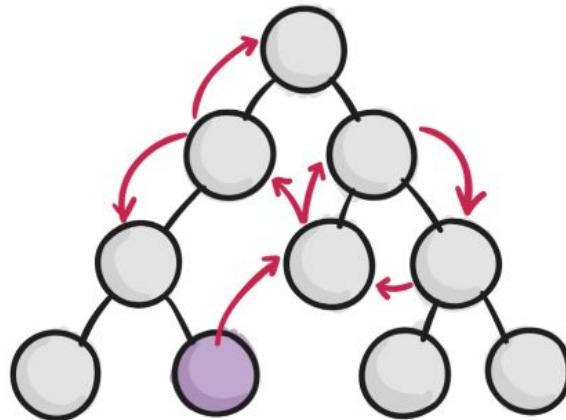
feature	uncontrolled	controlled
one-time value retrieval (e.g. on submit)	✓	✓
validating on submit	✓	✓
instant field validation	✗	✓
conditionally disabling submit button	✗	✓
enforcing input format	✗	✓
several inputs for one piece of data	✗	✓
dynamic inputs	✗	✓

{Redux}

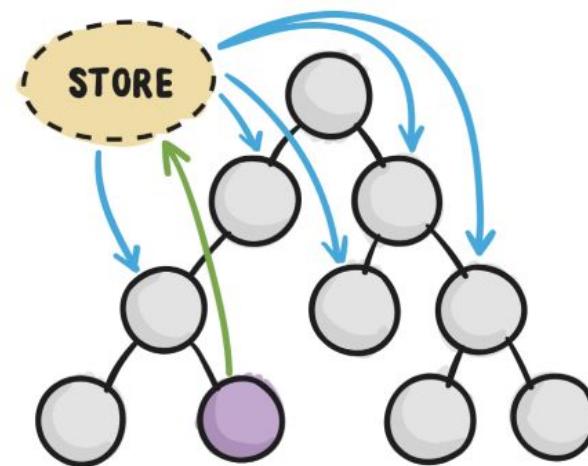
# **o que é Redux?**

**Redux simplifica a evolução de estados em uma aplicação quando há múltiplos estados para controlar e muitos componentes que precisam atualizar ou se inscrever nessa evolução, tirando a responsabilidade de cada componente de guardar o estado e passando para uma centralizada e única Store.**

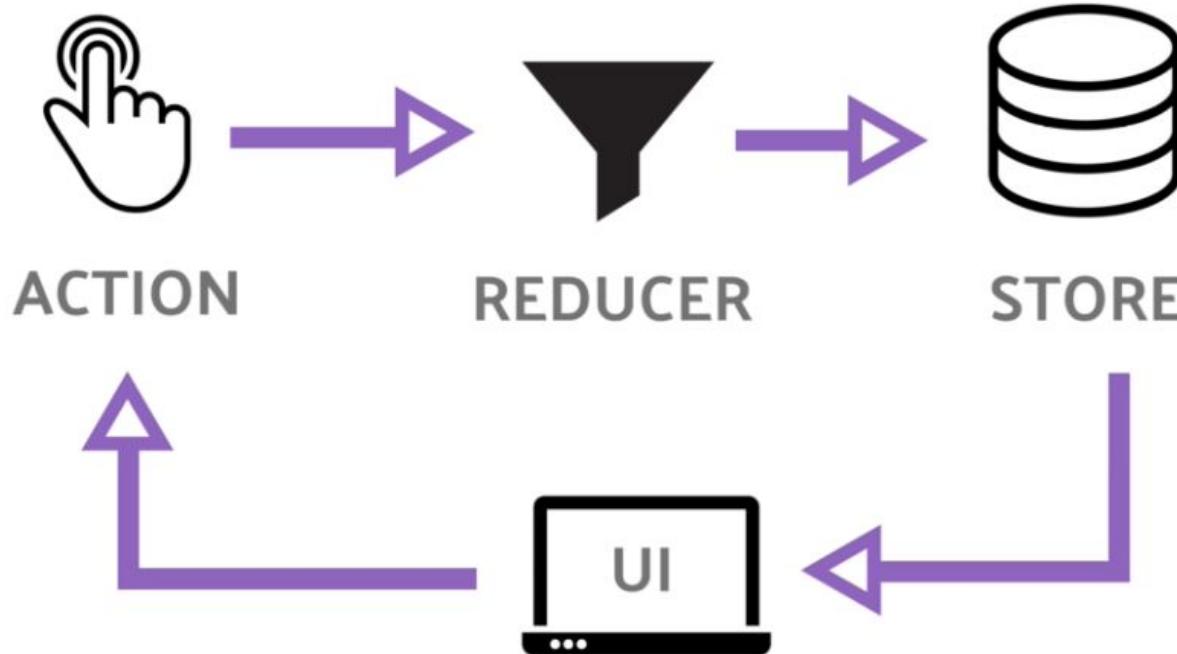
WITHOUT REDUX



WITH REDUX



● COMPONENT INITIATING CHANGE



# como usar?

- ❑ **actions**: objetos com dados que são enviados da sua aplicação para a Store. Elas são a única fonte de informação para a Store. Você manda eles para a store usando: store.dispatch().



```
export const toggleTodo = id => ({  
  type: 'TOGGLE_TODO',  
  id  
})
```

# como usar?

- ❑ **reducers:** eles executam a ação que altera o estado da aplicação em resposta a action enviada para a Store.



`(previousState, action) => nextState`



**Exercício**

# **o que vamos fazer?**

- ❑ **npx create-react-app create-react-redux-app**
- ❑ **create-react-app create-react-redux-app**
- ❑ **npm start**

# Ap

```
import React, { Component } from 'react';
import './App.css';
class App extends Component {
  render() {
    return (
      <div className="App" style={{ paddingTop: '10px' }}>
        <input type='text' />
        <button>
          Click me!
        </button>
        <h1>teste</h1>
      </div>
    );
  }
}
export default App;
```

**teste**

# Instalar o Redux?

❑ `npm i -D redux react-redux`



# src/store/Index.js

## ❑ **criar a store:**



```
import { createStore } from 'redux';
import { Reducers } from '../reducers';
export const Store = createStore(Reducers);
```

# src/reducers/index.js



```
import { clickReducer } from './clickReducer';
import { combineReducers } from 'redux';
export const Reducers = combineReducers({
  clickState: clickReducer,
});
```

# src/reducers/clickReducer.js



```
const initialState = {
  newValue: ''
};
export const clickReducer = (state = initialState, action) =>
{ switch (action.type) {
    case 'CLICK_UPDATE_VALUE':
        return {
            ...state,
            newValue: action.newValue
        };
    default:
        return state;
}
};
```

# src/actions/index.js



```
export const clickButton = value => ({
  type: 'CLICK_UPDATE_VALUE',
  newValue: value
});
```

# src/actions/actionsTypes.js



```
export const CLICK_UPDATE_VALUE = 'CLICK_UPDATE_VALUE';
```

# Alterar em src/actions/index.js e src/reducers/clickReducer.js



```
import { CLICK_UPDATE_VALUE } from  
'./actions/actionTypes';
```



```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import registerServiceWorker from './registerServiceWorker';
import { Provider } from 'react-redux';
import { Store } from './store';
ReactDOM.render(
  <Provider store={Store}>
    <App />
  </Provider>
, document.getElementById('root'));
registerServiceWorker();
```

# conectar o App.js com a store



```
const mapStateToProps = store => ({
  newValue: store.clickState.newValue
});
export default connect(mapStateToProps)
(App);
```

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import './App.css';
class App extends Component {
  render() {
    const { newValue } = this.props;
    return (
      <div className="App" style={{ paddingTop: '10px' }}>
        <input type='text' />
        <button>
          Click me!
        </button>
        <h1>{newValue}</h1>
      </div>
    );
  }
}
const mapStateToProps = store => ({
  newValue: store.clickState.newValue
});
export default connect(mapStateToProps)(App);
```

