Anotações do Curso ReactJS da B7Web - Módulo 03

(Por: Daniel Teixeira Quadros)

Sumário

Aula 01 – Conceito de Estado Global....2
Aula 02 – Redux: Introdução....2
Aula 03 – Redux: Criando Store e Reducers....3
Aula 04 – Redux: Connect e Dispatch....7
Aula 05 – Redux: Em Várias Telas....15
Aula 06 – Redux com Hooks....17
Aula 07 – Redux: Persistir Dados....20

Aula 01 – Conceito de Estado Global

Estado Global → Valores específicos que são acessíveis em todo o nosso sistema. (State global).

É muito útil no compartilhamento de informações e valores dentro do nosso sistema.

Vamos começar, a partir de agora a trabalhar na prática com o conceito de estado global, e para isto, existem diversos tipos de ferramentas e bibliotecas com vários tipos de procedimentos para tornar isto possível. Umas das principais formas (ferramentas) para isto é o **Redux** (biblioteca).

Aula 02 – Redux: Introdução

Para iniciar o módulo 3, iniciei um novo projeto React, para isto utilizei o comando **npx create-react-app nomedoprojeto**, dentro de uma nova pasta (modulo03/aula02). Esta etapa, está descrita nas minhas anotações do módulo 01, na aula 06.

Após criar um novo projeto, segui os passos que estão nas minhas anotações do módulo 02 na aula 01 que ensinam a fazer uma limpeza nos arquivos que inicialmente são criados na criação do React.

No terminal do VSCode, acessei a pasta do projeto recém criado com o comando **cd aula02** para dar prosseguimento nas instalações.

Segui também com a instalação do StyledComponentes utilizando o comando **npm install styled-components --save** que está explicado nas minhas anotações na aula 06 do módulo 02.

Ainda fiz a instalação do React Router com o comando **npm install react-router-dom** que está descrito nas minhas anotações da aula 21 do módulo 02.

Apenas após estas instalações que executei no terminal o **npm start** para começar a rodar este novo projeto.

Obs.: Não sei se a partir daqui, iremos utilizar os recursos do Styled Components ou do React Router Dom, mas achei interessante realizar todas as instalações que aprendi até aqui por questões de prática.

Agora, faremos a instalação de duas bibliotecas:

redux → Biblioteca do redux.
react-redux → Integração do react com o redux. (A biblioteca do redux não é feita especificamente para o react, apesar de ser um grande usuário da biblioteca).

Para isto utilizaremos o comando: **npm install redux react-redux**

```
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo03\aula02\aula02> npm install redux react-redux
npm WARN @babel/plugin-bugfix-v8-spread-parameters-in-optional-chaining@7.14.5 requires a peer of @babel/core@^7.13.0 but none is installed. You must instal
ll peer dependencies yourself.
npm WARN tsutils@3.21.0 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-dev || >= 3.6.0-b
eta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"
x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\watchpack-chokidar2\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":
"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\webpack-dev-server\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":
"x64"})

+ redux@4.1.0
+ react-redux@7.2.4
added 8 packages from 51 contributors and audited 1958 packages in 39.84s

147 packages are looking for funding
  run `npm fund` for details

found 3 moderate severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo03\aula02\aula02>
```

Funcionamento do Redux:

- Já que estamos trabalhando com um **estado global**, o Redux tem que estar acessível a toda a minha aplicação. Para isto, precisamos ter um local específico que abranja toda a aplicação.

Para utilizarmos o Redux, temos que aprender pelo menos três conceitos principais:

- Store → Local específico que conterá todos os dados do Redux.
- Reducers → Cada Reducer é responsável por armazenar várias informações sobre um determinado assunto. (Podemos ter diversos Reducers). Ex.: Um gerenciamento de usuário que queremos que esteja disponível em todas as minhas telas e que precisamos acessar as informações do usuário. Criamos um Reducer específico para os usuários e nesse Reducer armazenamos uma ou diversas informações sobre esses usuário. Então pegamos estas informações e jogamos dentro de Store.
- Actions → Ações que são criadas para manipular as informações que estão dentro de um determinado Reducer (Não alteramos as informações diretamente de dentro de um Reducer, utilizamos as Actions para isto).

Na teoria é assim que o Redux funciona: Temos a Store, dentro da Store, podemos ter várias Reducers e dentro da cada Reducer podemos ter diversas Actions.

No nosso componente, fazemos praticamente duas coisas: Lê informações do Reducer e executa ações do Reducer.

Aula 03 – Redux: Criando Store e Reducers

O primeiro passo aqui será criar o nosso Store.

Obs.: O Store deve ser criado antes de qualquer coisa, antes de começar a criar nosso aplicativo.
É importante lembrar que o nosso aplicativo começa no **index.js** por isto, **criaremos o nosso Store no index.js**.

No index.js iniciaremos importando dois recursos:

`import { Provider } from 'react-redux';` → É a “caixinha” que iremos utilizar. (Store) Que é utilizado para fazer a integração do Redux com o React.

`import { createStore } from 'redux';` →

`const store = createStore()` → Criamos aqui a variável store para armazenar as funcionalidades de createStore.

Abaixo, o index já vem inicialmente com a seguinte estrutura para o ReactDom:

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Porém, aqui, o **<React.StrictMode>** não é necessário, por isso excluiremos ele e deixaremos apenas o **<App/>**:

```
ReactDOM.render(
    <App />,
  document.getElementById('root')
);
```

Agora, colocamos o **<App/>** dentro do **<Provider>** que é a “caixinha” geral. Desta forma, tudo o que estiver no meu aplicativo (Dentro do Provider) terá acesso ao **store** e por consequência, terá acesso aos Reducers.

Além disto, precisamos adicionar ao Provider uma props que chamaremos de **store** e que será responsável por enviar a variável **store**. Ficamos com o ReactDOM inicialmente assim:

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

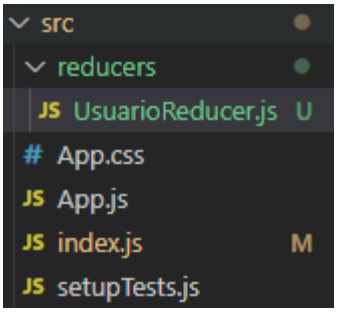
A partir deste momento precisamos criar o(s) Reducer(s).

Para isto, criaremos uma pasta que chamaremos de **reducers** dentro da pasta **src**. Nesta pasta armazenaremos todos os nossos reducers.

Criaremos um Reducer chamado de **UsuarioReducer.js**. Geralmente é utilizado a palavra Reducer na segunda parte do nome de um arquivo Reducer (Boas práticas).

Obs.: De acordo com o Bonieky, na documentação do Redux, é ensinado a utilizar um nome de arquivo com todas as letras em minúsculo.

Estrutura:



Vamos agora editar o UsuarioRedux.js

`const UsuarioReducer = (state = initialState, action) => {` → Criamos esta variável que contém uma função anônima que recebe dois parâmetros:

state → Tem duas funções. Na primeira vez que utilizarmos o Reducer ele virá com um estado inicial que nós mesmos iremos configuramos. Uma vez que este estado inicial for setado, podemos trocar as informações que tem lá.
Adicionamos ao state um estado inicial (**initialState** → Dados iniciais, os dados que começam o Reducer), Geralmente é utilizado um valor vazio, ou uma informação padrão.

action → Recebe a ação.

Criamos também a variável **initialState** com um objeto que contém o item **name** com uma String vazia:

```
const initialState = {
    name: ''
}
```

Dentro do UsuárioReducer, precisamos adicionar um **return**:

`return state` → Retornamos o próprio state, ou seja, quando ele rodar uma action ele vai receber o state como parâmetro, porém se não receber nada quer dizer que é a primeira vez que ele está rodando (por isto pega o estado inicial), e receberá uma action.

Acima do return, porém ainda dentro do UsuarioReducer é onde configuramos a ação que queremos fazer e depois retornamos o state alterado(atualizado) ou sem nenhum tipo de modificação.

Para isto, normalmente é utilizado um **switch**:

`switch(action.type) {` → Colocamos o **action.type** no switch, o .type define o tipo da ação.

Criamos um **case**:

`case 'SET_NAME':` → Estamos criando uma ação chamada de **SET_NAME** que será responsável por trocar o state.Geralmente é colocado tudo em maiúsculo, e se houver uma separação entre nomes, utilizamos o underline.

Aqui, utiilizamos um return diretamente dentro do **case**

`return {...state, name: action.payload.name}` → Aqui, copiamos o state já existente (**...state**) e alteramos o **name**. Alteramos o name utilizando o nome da action e algum valor específico (este valor também

está em action). Utilizamos o **payload** que é onde está as informações que queremos enviar, neste caso o name.

Nosso UsuarioReducer.js fica assim:

```
const initialState = {
    name: ''
}

const UsuarioReducer = (state = initialState, action) => {
    switch(action.type) {
        case 'SET_NAME':
            return {...state, name: action.payload.name}
            break
    }
    return state
}

export default UsuarioReducer
```

Para permitir que lá no **index.js** o nosso createStore que está sendo armazenado na variável **store** possa receber diversos Reducers e não apenas um, iremos criar a seguinte estrutura:

Dentro da pasta reducers, criamos outro arquivo também chamado de **index.js**(este nome facilita, porque quando puxamos a pasta ele já puxa este arquivo) e neste arquivo criaremos um item chamado **combineReducers** (Responsável por unir todos os Reducers):

```
import { combineReducers } from "redux";
```

Após ele, importamos para o mesmo arquivo todos os nossos Reducers. Que neste caso, até aqui, temos apena 1.:

```
import UsuarioReducer from "./UsuarioReducer";
```

Depois de importamos todos os nossos Reducers para cá, executamos o combineReducers já dentro do export. O combineReducers é uma função, e passamos como parâmetro um objeto com cada um dos meus Reducers:

```
export default combineReducers({
    usuario: UsuarioReducer
})
```

Aqui, chamamos o nosso Reducer de usuario. Porém, se quisermos, podemos enviar diretamente o nome UsuarioReducer:

```
export default combineReducers({
    UsuarioReducer
})
```

Porém, vamos utilizar este objeto da primeira maneira, dando um nome a ele. Desta forma, acessaremos o nosso Reducer utilizando o nome **usuario**.

O index.js da pasta reducers ficou assim:

```
import { combineReducers } from "redux";
import UsuarioReducer from "./UsuarioReducer";

export default combineReducers({
    usuario: UsuarioReducer
})
```

Para utilizarmos este Reducer, vamos retornar ao index.js da pasta src:

Criaremos um import que chamaremos de **Reducers** e aponteremos ele para a pasta **reducers**. Quando apontamos um import para uma pasta, ele automaticamente procura por um arquivo chamado **index**

```
import Reducers from './reducers';
```

Agora com o Reducers disponível, adicionamos ele como parâmetro da nossa variável **store**:

```
const store = createStore(Reducers)
```

Aula 04 – Redux: Connect e Dispatch

Obs.: Percebi, no final da Aula 03 que o Bonieky está aproveitando os arquivos da pasta **pages** e o arquivo **App.js** da pasta **src** do final do módulo 02. Por isto, copiei estes arquivos para o projeto que criei no início do módulo 03.

Nesta aula iremos aprender como conectar o Redux com a tela. Para isto vamos iniciar editando o nosso Home.js:

Vamos adicionar um import neste arquivo:

`import { connect } from 'react-redux'` → Recurso do Redux.

Vamos também alterar o export da Home:

`export default connect()(Home)` → Exportamos a função connect que enviará dois parâmetros que adicionaremos depois e no segundo parênteses, enviamos a Home.

Um destes parâmetros é uma variável que criaremos agora:

```
const mapStateToProps = (state) => {
    return {

    }
}
```

Esta variável, geralmente se utiliza este nome (**mapStateToProps**), contem uma função anônima que recebe um parâmetro chamado de **state** que vem do seu Reducer.

Retorna um objeto com alguns dados que pegamos do Reducer e podemos mostrar estes dados na tela como se fosse uma **prop**, por isso, adicionamos uma **prop** como parâmetro da **function Home**.

```
function Home(props){
```

Adicionamos também, no corpo da Página HOME um texto **NOME: ...** e duas quebras de linhas **

**

```
<div>
    <h4>Página HOME</h4>

    NOME: ...
    <br/><br/>
    <button onClick={handleButton}>Ir para a página SOBRE</button>
</div>
```

A informação que iremos passar através da prop da function Home virá do UsuarioReducer.js. Onde temos o UsuarioReducer.

Adicionaremos na variável initialState (src/reducers/UsuarioReducer.js) um valor para o item nome **Visitante**.

```
const initialState = {
    name: 'Visitante'
}
```

Feito isto, retornamos ao Home.js e adicionamos a seguinte informação no objeto de return do mapStateToProps:

```
name: state.usuario.name
```

→ Adicionamos um item chamado **name** que receberá do parâmetro **state** o item **usuario** (Que foi definido lá no nosso **combine** do arquivo **src/reducers/index.js**) que contem dentro dele a informação que queremos pegar, o **name** que vem do **UsuarioReducer.js**

Feito isto, podemos agora utilizar a variável **mapStateToProps** como primeiro parâmetro do connect:

```
const mapStateToProps = (state) => {
    return {
        name: state.usuario.name
    }
}

export default connect(mapStateToProps)(Home)
```

A partir daqui, já temos acesso a prop que vem lá do reducer:

```
function Home(props){
    let history = useHistory()
    const handleButton = () => {
        setTimeout(()=>{
            history.replace('/sobre')
        }, 2000)
    }
    return (
        <div>
            <h4>Página HOME</h4>

            NOME: {props.name}
            <br/><br/>
            <button onClick={handleButton}>Ir para a página SOBRE</button>
        </div>
    )
}
```

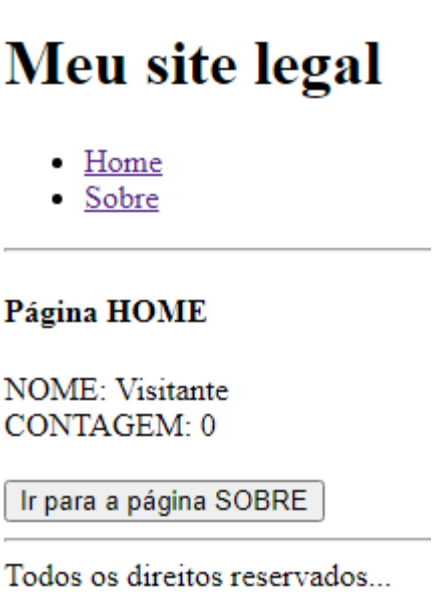
A partir daqui, iremos começar a manipular as informações que aparecerão na tela.

Vamos adicionar um contador na **initialState** do UsuarioReducer.js:

```
const initialState = {
    name: 'Visitante',
    contador: 0
}
```

E no Home.js vamos pegar esta informação:

```
const mapStateToProps = (state) => {
    return {
        name: state.usuario.name,
        contador: state.usuario.contador
    }
}
```



Já conseguimos ler as informações do Reducer, agora vamos aprender como executamos **actions** no **Reducer.**

Para isto, vamos criar o **segundo parâmetro** que iremos adicionar no **connect** no nosso **Home.js**:

`const mapDispatchToProps = (dispatch) => {` → Geramnte criamos um nome similar ao do primeiro parâmetro (o mapStateToProps). Este parâmetro recebe uma função anônima que receberá um parâmetro que chamaremos de **dispatch**

`return {` → Assim como no mapStateProps, **retornamos um objeto**, só que aqui, criaremos **funções específicas** que serão enviadas para o meu componente como se fosse uma prop.

`setName: (newName) => dispatch({` → Criamos uma função chamada **setName**, que será enviada para o meu componente como uma prop setName. Nesta função receberemos um novo nome que chamamos de **newName** (parâmetro) e retornamos diretamente aqui utilizando o **dispatch** que receberá como parâmetro um objeto que contém as duas informações que eu preciso lá no meu Reducer que são **type** e **payload**

`type: 'SET_NAME',` → type recebe o nome da ação que queremos executar (**SET_NAME**)

`payload: { name:newName}` → **payload** recebe um objeto com as informações que queremos enviar (**name:newName**)

Ficamos com o mapDispatchToProps assim:

```
const mapDispatchToProps = (dispatch) => {
    return {
        setName: (newName) => dispatch({
            type: 'SET_NAME',
            payload: { name:newName}
        })
    }
}
```

E adicionamos ele como segundo parâmetro do nosso connect:

```
export default connect(mapStateToProps, mapDispatchToProps)(Home)
```

Como resultado, temos agora disponível no nosso componente a função **SET_NAME** que recebe um parâmetro que enviamos para ela e ela roda a função **mapDispatchToProps** que vai executar **action setName** lá no nosso **Reducer**.

Para visualizarmos o funcionamento do mapDispatchToProps vamos fazer o seguinte:

No return da function Home vamos adicionar um Botão:

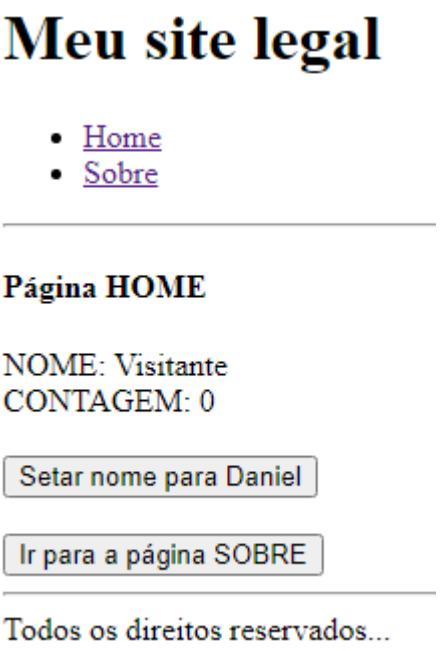
```
<button onClick={handleDaniel}>Setar nome para Daniel</button>
```

Que ao clicarmos nele executará a seguinte função:

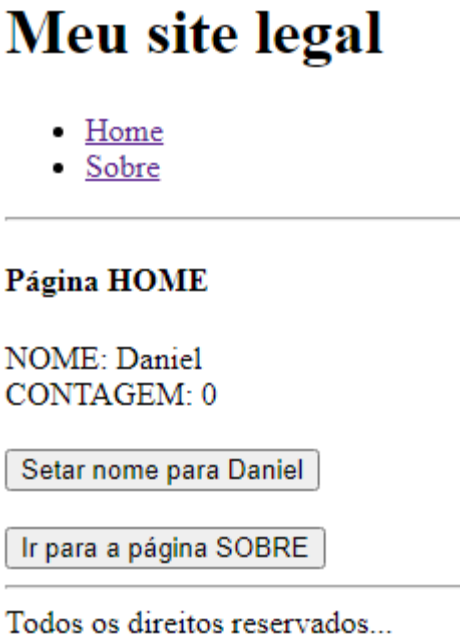
```
const handleDaniel = () => {
    props.setName('Daniel')
}
```

Esta função executa o **setName** e altera o seu conteúdo para a String **'Daniel'**

Agora a nossa página está assim:



Porém, se clicarmos no botão **Setar nome para Daniel**, o nome Visitande é substituido pelo nome Daniel:



Vamos criar mais um botão para explorar um pouco mais este recurso:

No return da function Home adicionamo um botão que chamamos de **+1**:

```
<button onClick={handleIncrement}>+1</button>
```

Acima, criamos a função **handleIncrement** que não recebe nenhum parâmetro:

```
const handleIncrement = () => {
    props.increment()
}
```

E no **mapDispatchToProps** adicionamos o **increment** com uma ação que criaremos no nosso **Reducer** e chamaremos de **INCREMENT_CONTADOR**

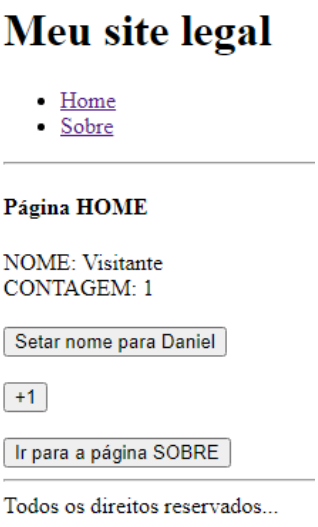
```
return {
    setName: (newName) => dispatch({
        type: 'SET_NAME',
        payload: { name:newName}
    }),
    increment: () => dispatch({
        type: 'INCREMENT_CONTADOR'
    })
}
```

Adicionamos o **INCREMENT_CONTADOR** no UsuarioReducer:

```
const UsuarioReducer = (state = initialState, action) => {
    switch(action.type) {
        case 'SET_NAME':
            return {...state, name: action.payload.name}
            break
        case 'INCREMENT_CONTADOR':
            let newCount = state.contador + 1
            return { ...state, contador:newCount }
            break
    }
    return state
}

export default UsuarioReducer
```

Com estas alterações, o nosso site ficou assim:

Se clicarmos no botão **+1** o valor do contador é alterado:



Terminamos a aula com o UsuarioReducer.js assim:

```
const initialState = {
    name: 'Visitante',
    contador: 0
}

const UsuarioReducer = (state = initialState, action) => {
    switch(action.type) {
        case 'SET_NAME':
            return {...state, name: action.payload.name}
            break
        case 'INCREMENT_CONTADOR':
            let newCount = state.contador + 1
            return { ...state, contador:newCount }
            break
    }
    return state
}

export default UsuarioReducer
```

E com o Home.js assim:

```
import React from 'react'
import {useHistory} from 'react-router-dom'
import { connect } from 'react-redux'

function Home(props){
    let history = useHistory()
    const handleButton = () => {
        setTimeout(()=>{
            history.replace('/sobre')
        }, 2000)
    }

    const handleDaniel = () => {
        props.setName('Daniel')
    }

    const handleIncrement = () => {
        props.increment()
    }
```

```
20
21      return (
22          <div>
23              <h4>Página HOME</h4>
24
25              NOME: {props.name}<br/>
26              CONTAGEM: {props.contador}<br/><br/>
27              <button onClick={handleDaniel}>Setar nome para Daniel</button>
28              <br/><br/>
29              <button onClick={handleIncrement}>+1</button>
30              <br/><br/>
31              <button onClick={handleButton}>Ir para a página SOBRE</button>
32
33          </div>
34      )
35  }
36
```

```
37  const mapStateToProps = (state) => {
38      return {
39          name: state.usuario.name,
40          contador: state.usuario.contador
41      }
42  }
43
44  const mapDispatchToProps = (dispatch) => {
45      return {
46          setName: (newName) => dispatch({
47              type: 'SET_NAME',
48              payload: { name:newName}
49          }),
50          increment: () => dispatch({
51              type: 'INCREMENT_CONTADOR'
52          })
53      }
54  }
55
56  export default connect(mapStateToProps, mapDispatchToProps)(Home)
```

Aula 05 – Redux: Em Várias Telas

É importante entendermos que em todas as telas que quisermos pegar informações ou manipular informações precisaremos utilizar o **connect** que vimos na aula passada. Também veremos como fazer para não perdermos as alterações realizadas em uma tela quando mudamos de tela no nosso sistema

Para iniciarmos esta aula, vamos utilizar a página **Login.js** lembrando que tornamos a página **Sobre.js** privada com o **PrivateRoute** do **App.js**. Ou seja, a página Sobre só será acessada quando a variável **isLogged** do **App.js** estiver com a condição **true.**

Criamos na página **Login.js** a mesma estrutura que criamos na página **Home.js** da aula passada:

```
1   import React from 'react'
2   import { connect } from 'react-redux'
3
4   function Login(props){
5
6       const handleLisi = () => {
7           props.setName('Lisi')
8       }
9
10      return (
11          <div>
12              <h4>Página Login</h4>
13              O nome é: {props.name}
14              <br/><br/>
15              <button onClick={handleLisi}>troca nome para Lisi</button>
16          </div>
17      )
18  }
19
20  const mapStateToProps = state => ({
21      name: state.usuario.name
22  })
23
24  const mapDispatchToProp = dispatch => ({
25      setName: (newName) => dispatch({
26          type: 'SET_NAME',
27          payload: {name: newName}
28      })
29  })
30
31  export default connect(mapStateToProps, mapDispatchToProp)(Login)
```

Obsesvações:

- Adicionamos o **import connect** na linha 2.
- Adicionamos a **props** na **function Login.**
- Criamos a variável **handleLisi** que setou o nome **Lisi** em name através da **props.**
- No return da function Login, adicionamos o texto '**O nome é:**' e a **{props.name}**.
- Ainda no return da function Login adicionamos o botão que aciona a function da variável **handleLisi** que é responsável por alterar o nome para 'Lisi'.
- Abaixo da function Login, criamos então as variáveis que armazenam as function **state** e **dispatch**.

Obs.: Aqui, para estas duas funções, utilizamos a estrutura mais resumida para uma função com um único retorno. A estrutura que estávamos usando é a seguinte:

```
const mapStateToProps = (state) => {
    return{
        name: state.usuario.name
    }
}
```

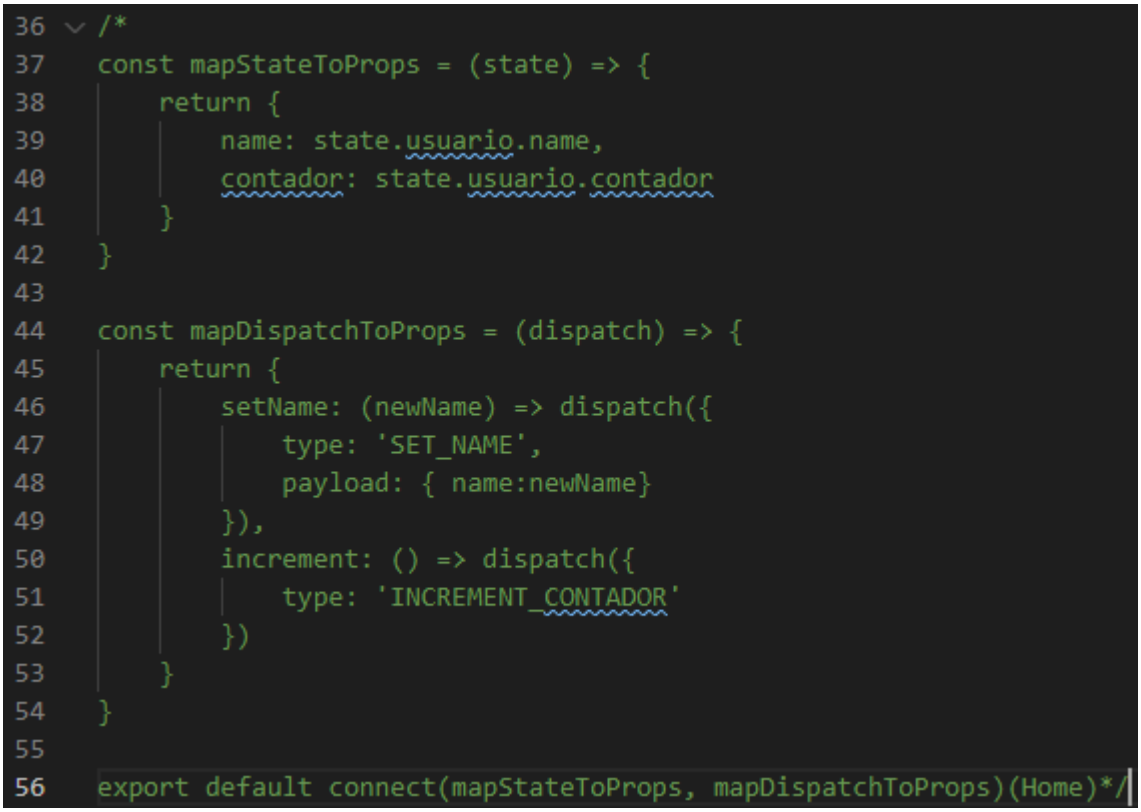
- E no export (última linha), adicionamos a estrutura do connect

Aula 06 – Redux com Hooks

Até aqui, apredemos a utilizar o Redux de uma forma mais difícil, porém foi importante entender a estrutura desta forma porque era a forma que se trabalhava anteriormente e podemos nos deparar com sistemas mais antigos que utilizam esta estrutura.

A partir de agora, aprenderemos a trabalhar com o Redux de uma forma mais atual e mais fácil, que é utilizando **Hooks**.

Começaremos editando a nossa página Home. Comaçamos removendo todo o conteúdo abaixo. (No momento achei melhor apenas comentar o código):



```
/*
const mapStateToProps = (state) => {
    return {
        name: state.usuario.name,
        contador: state.usuario.contador
    }
}

const mapDispatchToProps = (dispatch) => {
    return {
        setName: (newName) => dispatch({
            type: 'SET_NAME',
            payload: { name:newName}
        }),
        increment: () => dispatch({
            type: 'INCREMENT_CONTADOR'
        })
    }
}

export default connect(mapStateToProps, mapDispatchToProps)(Home)*/
```

Ao final do código, apenas adicionamos o nosso export do Home padrão:

```
export default Home
```

Substituímos o nosso import do connect (`import { connect } from 'react-redux'`) por outro import, o useSelector (`import { useSelector } from 'react-redux'`) que é uma biblioteca do react-redux.

Usaremos o useSelector da seguinte forma:

Ao invés da função mapStateToProps que excluímos (comentamos) vamos utilizar duas variáveis que criaremos agora

`const name = useSelector(state => state.usuario.name)` → Criamos aqui a variável **name** e atribuímos a ela a função **useSelector** e dentro desta função utilizaremos uma função anônima que atribui à state o state.usuario.name que trás o name do usuarioReducer.

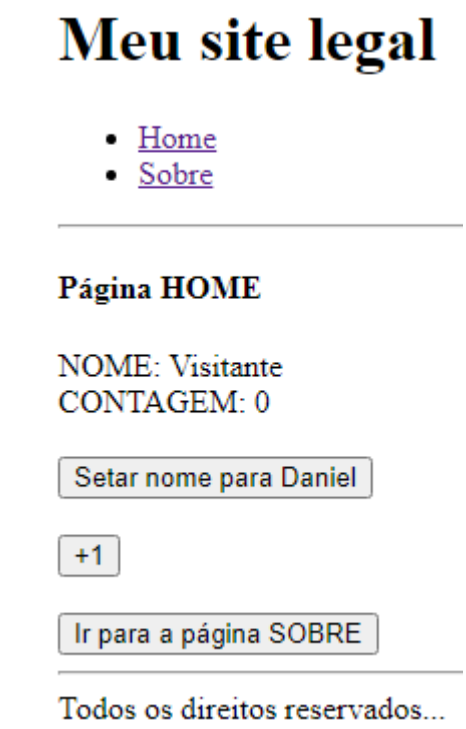
Desta forma, no return da function Home, utilizamos este name, removendo a props (`NOME: {props.name}<br/>`) e ficando apenas assim: `NOME: {name}<br/>`

Com o contador, fazemos o mesmo procedimento:

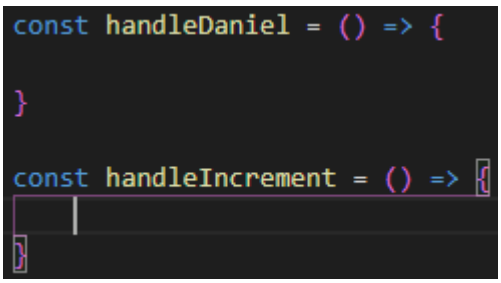
`const contador = useSelector(state => state.usuario.contador)` → Criamos aqui a variável **contador** e atribuímos a ela a função **useSelector** e dentro desta função utilizaremos uma função anônima que atribui à state o state.usuario.contador que trás o contador do usuarioReducer.

E também, no return da function Home, utilizamos este contador, removendo a props (`CONTAGEM: {props.contador}<br/><br/>`) e ficando apenas assim: `CONTAGEM: {contador}<br/><br/>`

Observamos que com esta nova estrutura, a nossa aplicação responde corretamente:



Não utilizaremos mais a props da function Home `function Home(){`
e com isto, excluiremos as props das variáveis handleDaniel (`props.setName('Daniel')`) e handleIncrement (`props.increment()`) Ficando com ambas assim:



Vamos agora criar o nosso processo de **dispatch** ou seja, o processo de rodar uma **action** lá no meu **reducer** Para isto, utilizaremos no nosso import, outro componente chamado de **useDispatch**.

```
import { useSelector, useDispatch } from 'react-redux'
```

Colocaremos o **useDispatch** em uma variável que chamaremos de **dispatch**:

```
const dispatch = useDispatch()
```

E, a partir de agora, sempre que quisermos utilizar os recursos do **useDispatch**, utilizaremos eles através da nossa variável **dispatch**.

Aqui, na nossa aplicação, utilizaremos o dispatch primeiramente na variável **handleDaniel** da seguinte forma:

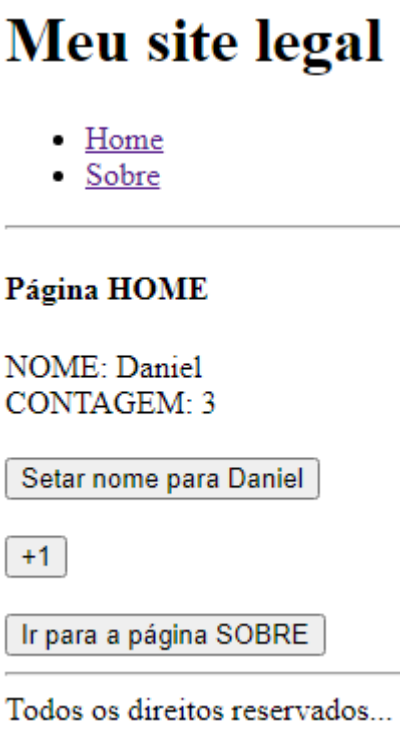
```
const handleDaniel = () => {
    dispatch({
        type: 'SET_NAME',
        payload: { name: 'Daniel'}
    })
}
```

Através da função anônima, acionamos o dispatch einserimos um objeto que altera no name (`type: 'SET_NAME',`) para Daniel (`payload: { name: 'Daniel'}`)

E na variável **handleIncrement** utilizaremos o **dispatch** da mesma forma:

```
const handleIncrement = () => {
    dispatch({
        type: 'INCREMENT_CONTADOR'
    })
}
```

A partir daqui, o nosso sistema, está manipulando as informações normalmente com esta nova estrutura, da mesma forma que estava funcionando anteriormente com a estrutura antiga:



Aula 07 – Redux: Persistir Dados

Persistir os dados → Mesmo após a atualização da tela ou mesmo fechar a aplicação, as alterações realizadas permanecerão. Isto porque, utilizaremos o **local storage** para salvar as informações.

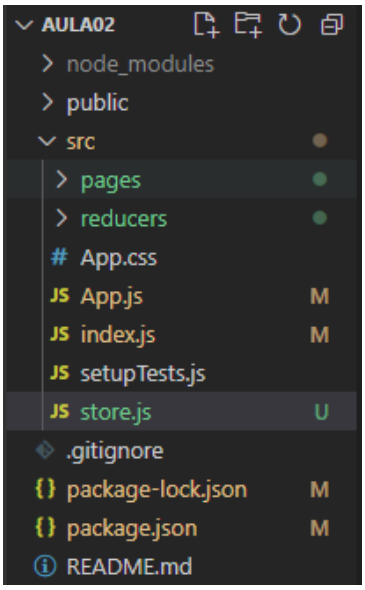
Para isto instalaremos uma biblioteca chamada de **redux persist**

npm install redux-persist → Comando utilizado no terminal para a instalação da biblioteca redux persist.

```
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo03\aula02\aula02> npm install redux-persist
npm WARN @babel/plugin-bugfix-v8-spread-parameters-in-optional-chaining@7.14.5 requires a peer of @babel/core@^7.13.0 but none is installed
. You must install peer dependencies yourself.
npm WARN tsutils@3.21.0 requires a peer of typescript@>=2.8.0 || >= 3.2.0-dev || >= 3.3.0-dev || >= 3.4.0-dev || >= 3.5.0-dev || >= 3.6.0-d
ev || >= 3.6.0-beta || >= 3.7.0-dev || >= 3.7.0-beta but none is installed. You must install peer dependencies yourself.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":
"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\watchpack-chokidar2\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os"
:"win32","arch":"x64"})
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.13 (node_modules\webpack-dev-server\node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.13: wanted {"os":"darwin","arch":"any"} (current: {"os"
:"win32","arch":"x64"})

+ redux-persist@6.0.0
added 1 package and audited 1960 packages in 38.101s

147 packages are looking for funding
  run `npm fund` for details

found 3 moderate severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
PS D:\Informática\Programação\2021\Daniel\B7Web\ReactJS\modulo03\aula02\aula02>
```

Onde aplicamos o redux persist?

Não precisamos fazer nenhuma alteração na nossa aplicação, a única alteração que precisamos fazer é no nosso **src/index.js**. Porém antes de fazermos qualquer alteração aqui, vamos criar um novo arquivo na pasta **src** que chamaremos de **store.js**:



Iniciaremos o nosso store.js importando os recursos necessários:

```
src > JS store.js
1  import React from 'react'
2  import { createStore } from 'redux'
3  import { persistStore, persistReducer } from 'redux-persist'
4  import storage from 'redux-persist/lib/storage'
5
6  import Reducers from './reducers'
```

import { persistStore, persistReducer } from 'redux-persist' → recursos do redux persist.
import storage from 'redux-persist/lib/storage' → Informa que é necessário do storage para realizar este procedimento.
import Reducers from './reducers' → Importamos os nossos Reducers.

```
8   const persistConfig = {
9       key: 'root',
10      storage,
11      whitelist: ['usuario']
```

const persistConfig = { → Objeto com algumas configurações.
key: 'root', → key é obrigatório, funciona como uma chave.
Storage, → storage também é obrigatório → Este item pode ser utilizado na forma storage: storage. Que trás os recursos do import do storage.
whitelist: ['usuario'] → whitelist é opcional. Adicionamos a ele um array com os nomes dos reducers que queremos que sejam salvos (mantidos).

```
const pReducer = persistReducer(persistConfig, Reducers)

const store = createStore(pReducer)
const persistor = persistStore(store)

export { store, persistor }
```

`const pReducer = persistReducer(persistConfig, Reducers)` → pReducer é o nome da variável que criamos aqui que recebe a função persistReducer que importamos do redux-persist e enviamos para ela dois parâmetros, as informações da variável persistConfig que criamos aqui e os nossos Reducers.

`const store = createStore(pReducer)` → Criamos a variável store que recebe os recursos do createStore que importamos do redux e enviamos como parâmetro o pReducer.

`const persistor = persistStore(store)` → Criamos a variável persistor e salvamos nela os recursos do persistorStore que importamos do redux-persist e enviamos como parâmetro a variável store que criamos acima.

`export { store, persistor }` → No final do arquivo, não utilizamos o export default, ao invés disto, exportamos apenas as nossas variáveis **store** e **persistor**.

Ficamos com o nosso **store.js** assim:

```
src > JS store.js > ...
1  import React from 'react'
2  import { createStore } from 'redux'
3  import { persistStore, persistReducer } from 'redux-persist'
4  import storage from 'redux-persist/lib/storage'
5
6  import Reducers from './reducers'
7
8  const persistConfig = {
9      key: 'root',
10     storage,
11     whitelist: ['usuario']
12 }
13
14 const pReducer = persistReducer(persistConfig, Reducers)
15
16 const store = createStore(pReducer)
17 const persistor = persistStore(store)
18
19 export { store, persistor }
```

Com o nosso store.js criado, vamos reestruturar o **src/index.js** da seguinte maneira:

Excluímos as seguintes linhas:

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

E adicionamos o import do **store.js**:

```
import { store, persistor } from './store'
```

Importamos também outro recurso do **redux-persist**, o **persistGate**:

```
import { PersistGate } from 'redux-persist/integration/react'
```

Adicionamos o persistGate dentro do **Provider** e o **App** dentro do **PersistGate**

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

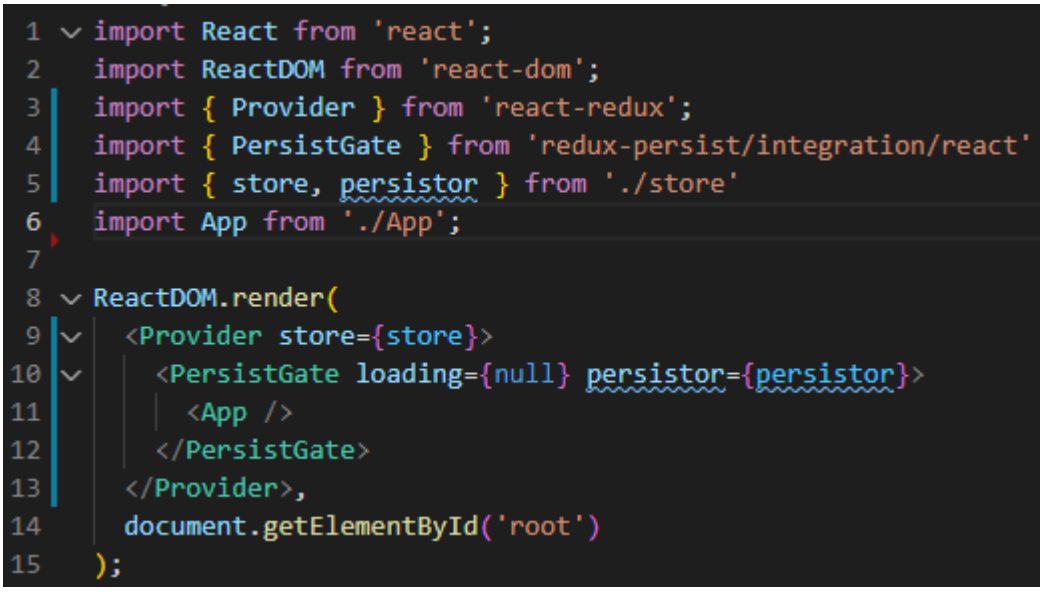
PersisteGate → É utilizado para garantir que a aplicação irá rodar apenas após que todo o conteúdo do Storage for carregado para o Redux.

Ainda, Com o carregamento do PersistGate, enviamos duas props:

```
<PersistGate loading={null} persistor={persistor}>
```

loading={null} → Carregamento. É utilizado para evitar que a tela seja carregada sem informações.
persistor={persistor} → Enviamos o nosso **persistor**.

Nosso **src/index.js** ficou assim:



```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { PersistGate } from 'redux-persist/integration/react'
import { store, persistor } from './store'
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <PersistGate loading={null} persistor={persistor}>
      <App />
    </PersistGate>
  </Provider>,
  document.getElementById('root')
);
```

A partir deste momento, a função do persist já está funcionando nosso sistema, ou seja, quando manipulamos as informações da nossa aplicação, mesmo após a atualização de página, as alterações realizadas se mantém.