ReactJS
Redux

1 Introdução

A documentação oficial do Redux o define como

"Um contêiner de estado previsível para aplicações Javascript"

Veja o Link 1.1.

Link 1.1
https://redux.js.org/

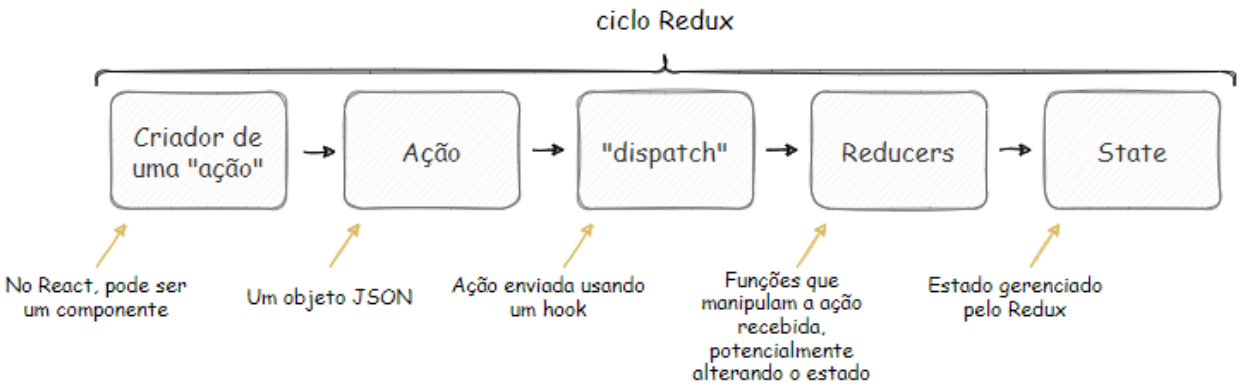
Vale destacar os seguintes pontos da documentação.

- Trata-se de uma biblioteca para manipulação de **estado centralizado**.

- Embora seja muito utilizado em aplicações React, não foi produzido com esse único propósito.

1.1 (Ciclo básico do Redux) O Redux possui um ciclo composto por algumas partes que levam um nome um tanto específico. Veja a Figura 1.1.1.

Figura 1.1.1

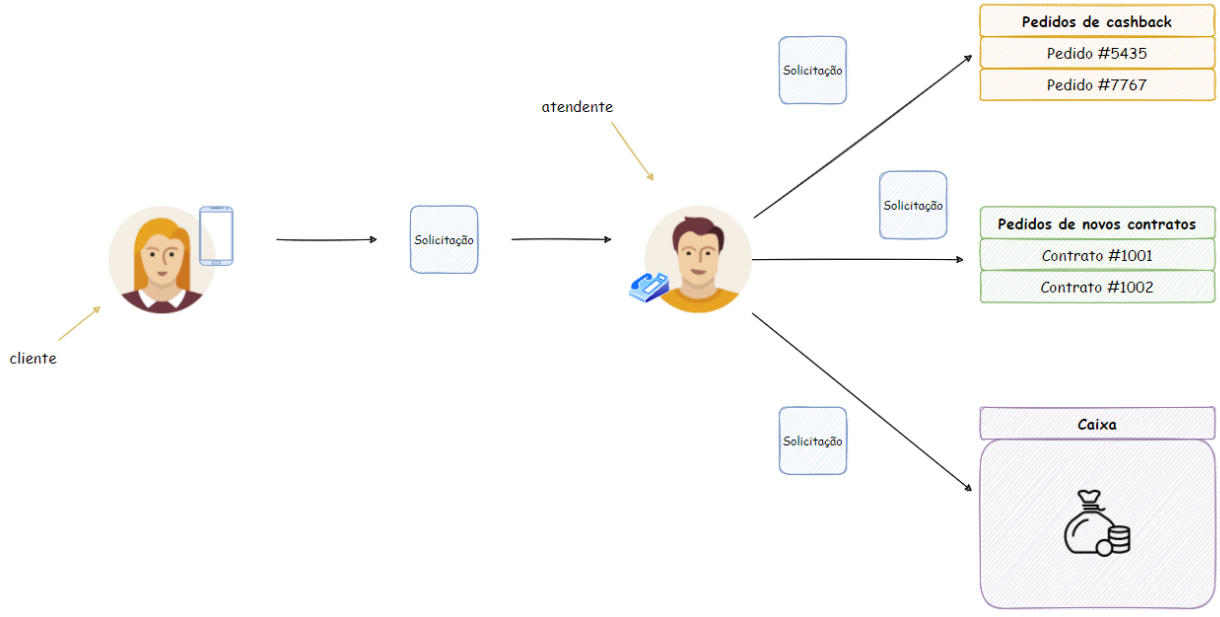


1.2 (Cashback oferecido por uma empresa de cartões de crédito: uma analogia) Para simplificar o entendimento de cada item do ciclo ilustrado, considere a seguinte analogia.

I. Uma empresa de cartão de crédito oferece "**cashback**" para compras feitas utilizando seus cartões.
II. Pessoas podem adquirir seus cartões de crédito. Para isso, assinam um **contrato** com a empresa. Elas pagam uma **taxa** por isso.
III. Após acumular uma quantidade de compras, os clientes podem realizar **pedidos** para obter seu cashback.
IV. A empresa paga os pedidos em **dinheiro**.
V. A empresa armazena o **histórico de pedidos de cashback**.
VI. A empresa armazena o **histórico de contratos assinados**.
VII. A empresa possui um **caixa** a partir do qual os pagamentos acontecem.

Veja a Figura 1.2.1.

Figura 1.2.1

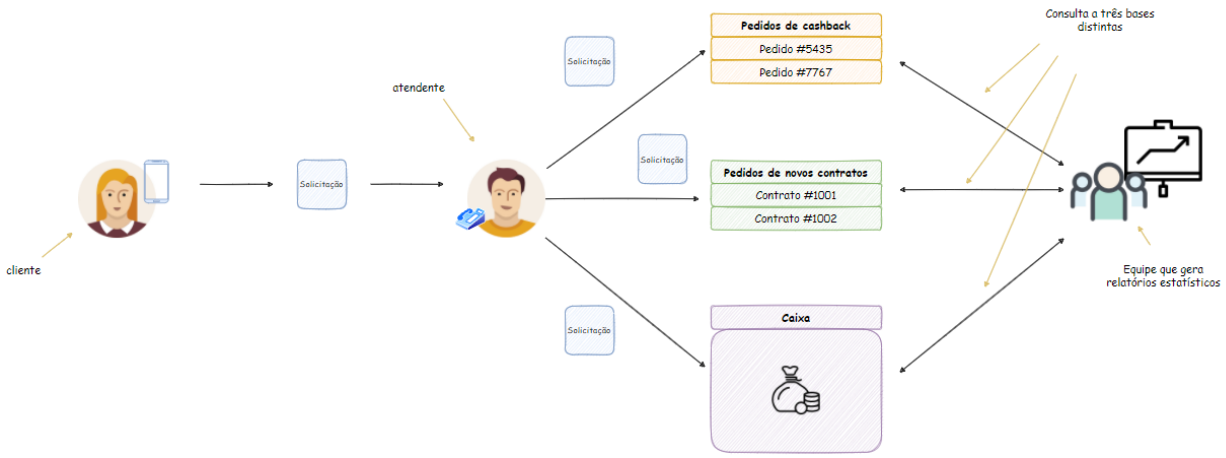


Na Figura 1.2.1, o que ocorre é o seguinte:

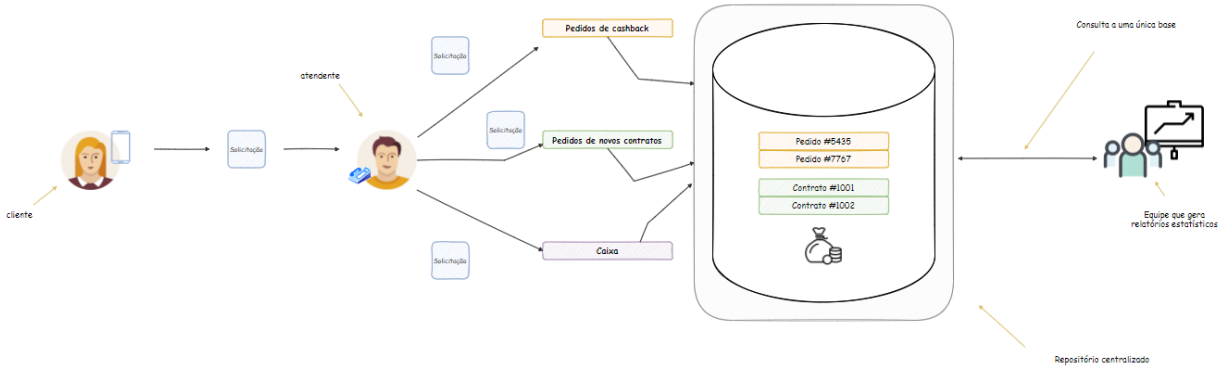
I. Um cliente tem uma solicitação. Ela pode ser de tipos diversos.
II. O cliente entrega a sua solicitação a um atendente.
III. O atendente entrega uma cópia da solicitação **para cada** departamento da empresa.
IV. Cada departamento que recebe a solicitação decide como manipulá-la.
V. Dependendo do tipo da solicitação, um departamento pode decidir ignorá-la. Por exemplo, o departamento de pedidos de cashback não está interessado em solicitações de novos contratos.

1.3 (Uma equipe deseja desenvolver relatórios estatísticos) Suponha que há uma equipe nesta empresa que deseja desenvolver relatórios estatísticos sobre os contratos, pedidos de cashback e fluxo de caixa. Para tal, seria necessário consultar as bases de cada um dos departamentos a fim de obter os dados de interesse. Veja a Figura 1.3.1

Figura 1.3.1



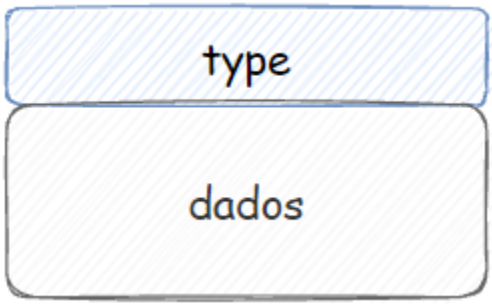
A Figura 1.3.2 mostra uma alternativa: armazenar os dados de todos os departamentos em um único **repositório centralizado**.



1.4 (Com o que se parece uma solicitação?) Neste exemplo, teremos três tipos de solicitações:

- Criação de novo contrato. Vamos supor que há uma taxa inicial a ser paga pelo contrato. Além disso, a pessoa interessada deve informar o seu nome.
- Pedido de cashback. Uma pessoa informa seu nome e o valor que deseja obter como cashback.
- Cancelamento de contrato. Uma pessoa pode cancelar seu contrato a qualquer momento. Para tal, ela informa seu nome. A Figura 1.4.1 mostra a estrutura básica de uma solicitação.

Figura 1.4.1



Nota. O nome "type" é obrigatório. As demais partes são decididas pelo desenvolvedor. Veja o que diz a documentação.

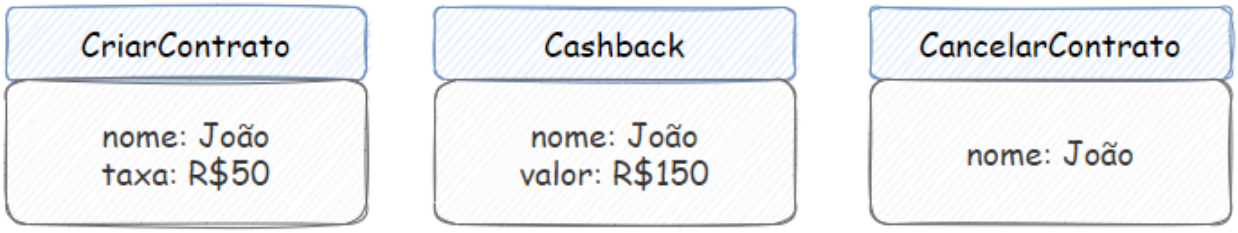
"Actions must have a 'type' field that indicates the type of action being performed. Types can be defined as constants and imported from another module. It's better to use strings for type than Symbols because strings are serializable. Other than type, the structure of an action object is really up to you."

Veja mais na página acessível por meio do Link 1.4.1.

Link 1.4.1
https://redux.js.org/api/store

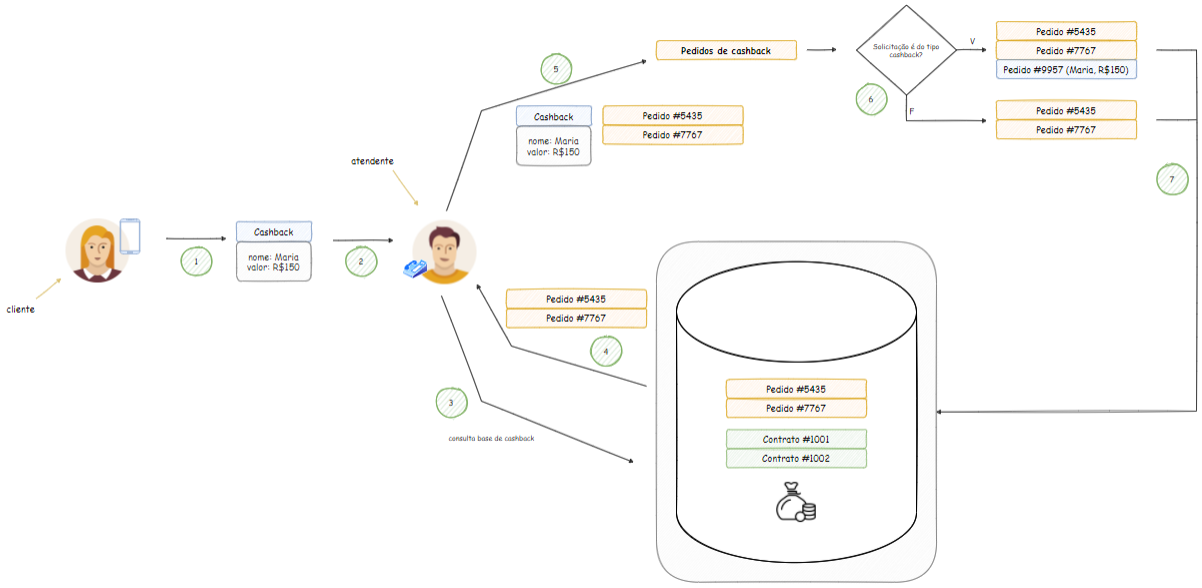
A Figura 1.4.2 mostra os três tipos de solicitações que nos são de interesse.

Figura 1.4.2



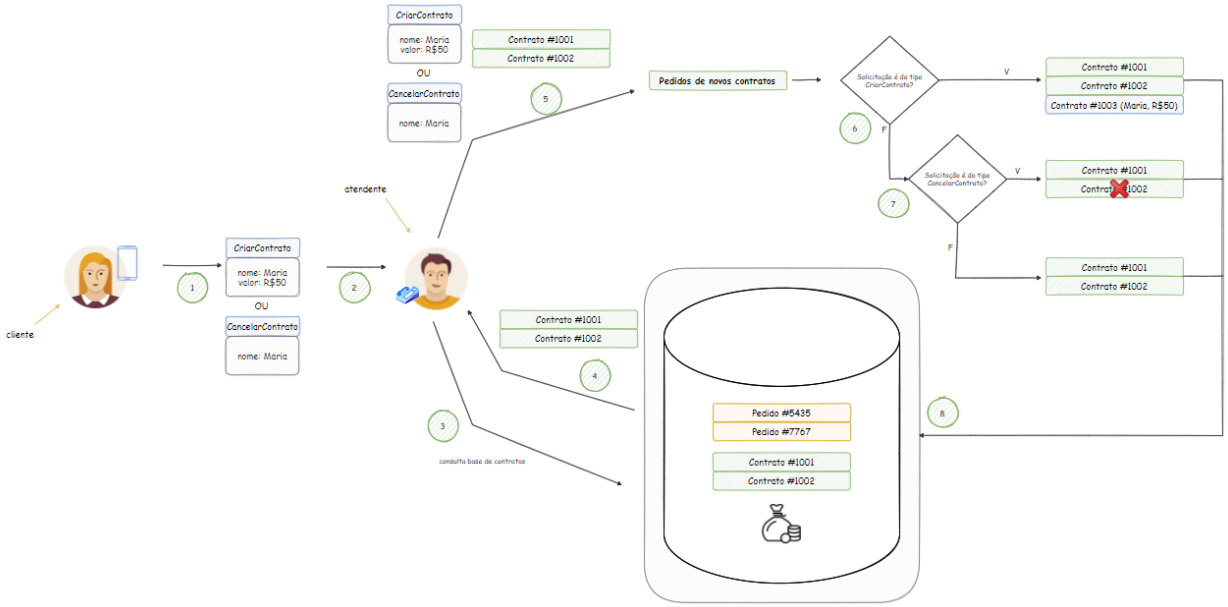
1.5 (Detalhes sobre um pedido de cashback: funções do atendente e o departamento de "Pedidos de cashback") Como vimos, cabe ao atendente receber as solicitações e direcioná-las ao departamento adequado. Dada a existência da base centralizada, caberá a ele, também, consultá-la previamente e entregá-la ao departamento alvo da solicitação. Por exemplo, quando ele recebe uma solicitação de cashback, ele faz uma consulta à base e obtém todos os registros referentes a cashback. Depois disso, entrega a lista de registros de cashback e a nova solicitação ao departamento de pedidos de cashback que, por sua vez, decide se a base centralizada deve ou não ser atualizada. Veja a Figura 1.5.1.

Figura 1.5.1


cliente
Cashback
nome: Maria
valor: R$150
1
2
atendente
3
consulta base de cashback
Pedido #5435
Pedido #7767
4
5
Pedidos de cashback
Cashback
nome: Maria
valor: R$150
Pedido #5435
Pedido #7767
Solicitação é do tipo cashback?
V
F
6
Pedido #5435
Pedido #7767
Pedido #9957 (Maria, R$150)
Pedido #5435
Pedido #7767
7
Pedido #5435
Pedido #7767
Contrato #1001
Contrato #1002

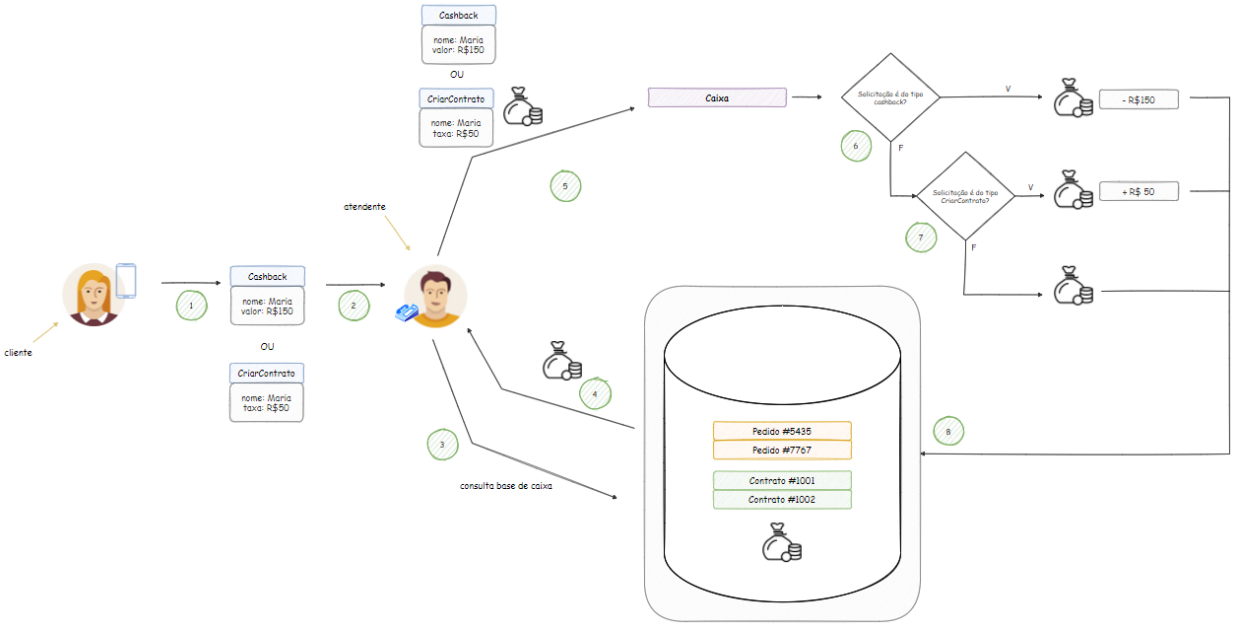
1.6 (Detalhes sobre um pedido de novo contrato: funções do atendente e o departamento de "contratos") Ao receber solicitações dos tipos NovoContrato ou CancelarContrato, o atendente consulta a base de contratos existentes e a entrega ao departamento de novos contratos, incluindo a nova solicitação. Veja a Figura 1.6.1.

Figura 1.6.1



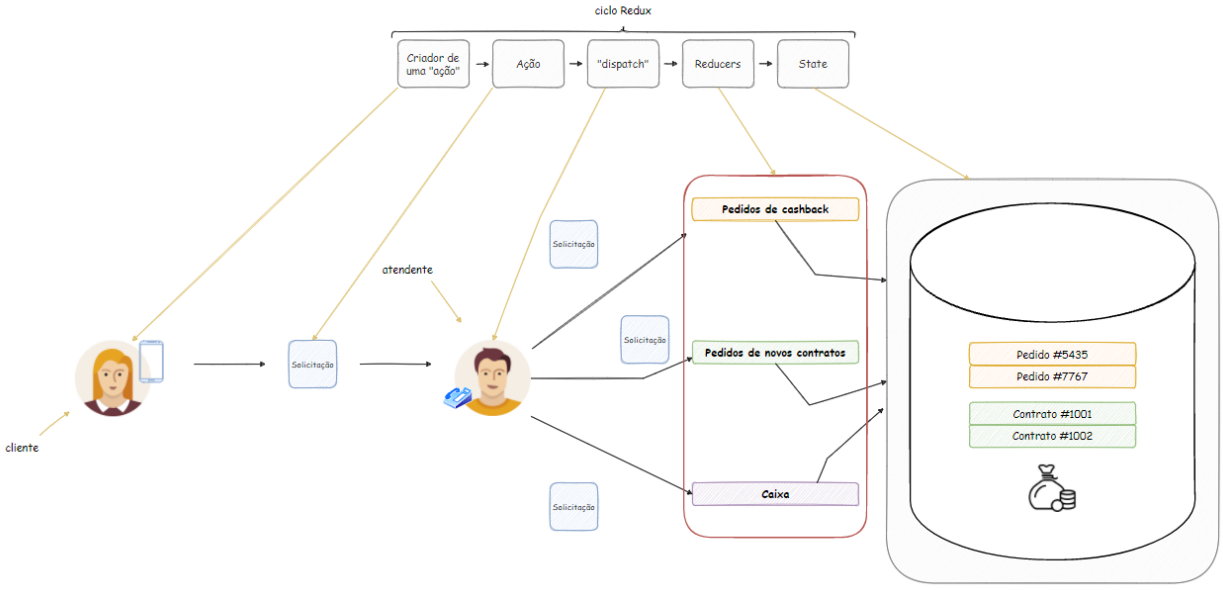
1.7 (Detalhes de um pedido por cashback ou novo contrato: funções do atendente e o departamento "caixa") Pedidos de novos contratos e por cashback também são de interesse para o departamento "caixa". Veja a Figura 1.7.1.

Figura 1.7.1



1.8 (Associando os nomes do ciclo Redux com as partes retratadas na simulação da empresa de cashback) Cada parte retratada nas simulações feitas até então está associada a um conceito do ciclo Redux. Veja a Figura 1.8.1.

Figura 1.8.1



2 Desenvolvimento

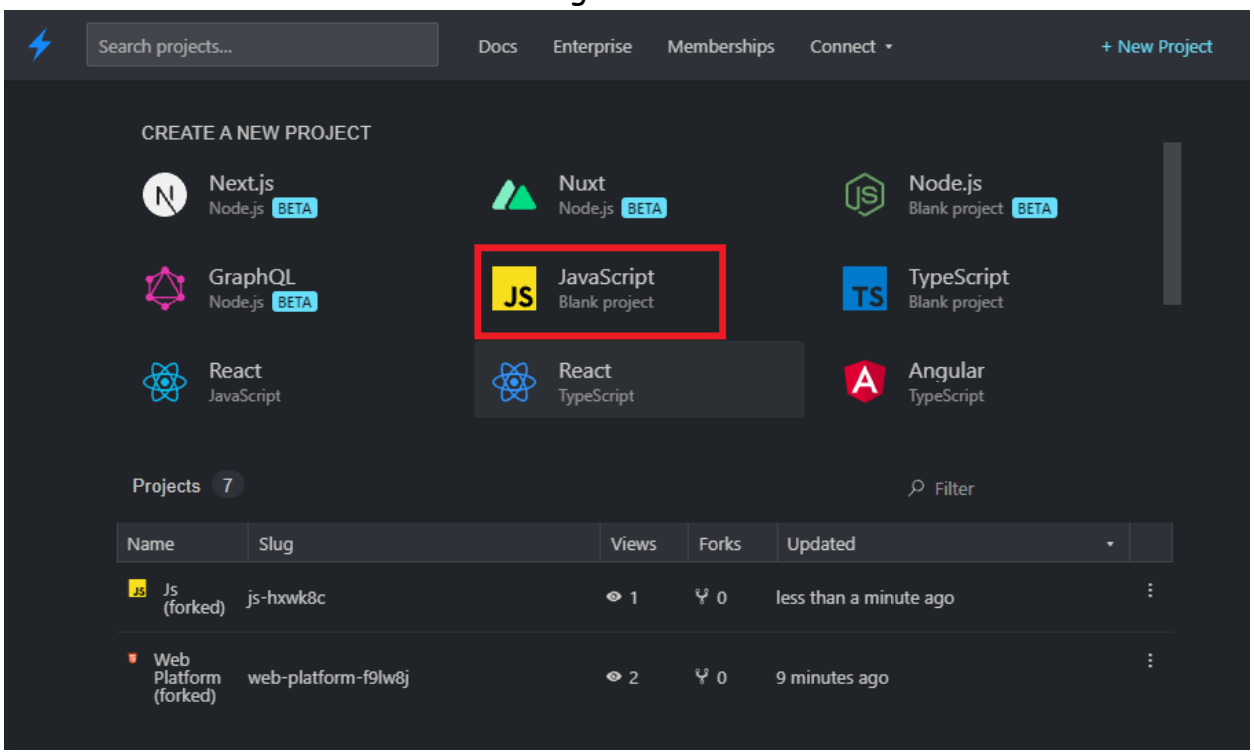
Nesta seção, implementaremos as funcionalidades da empresa descrita. Para tal, utilizaremos o ambiente StackBlitz. Encontre a sua página oficial por meio do Link 2.1.

Link 2.1
https://stackblitz.com/

Pode ser de interesse fazer login com o seu Github, assim você poderá armazenar seus projetos.

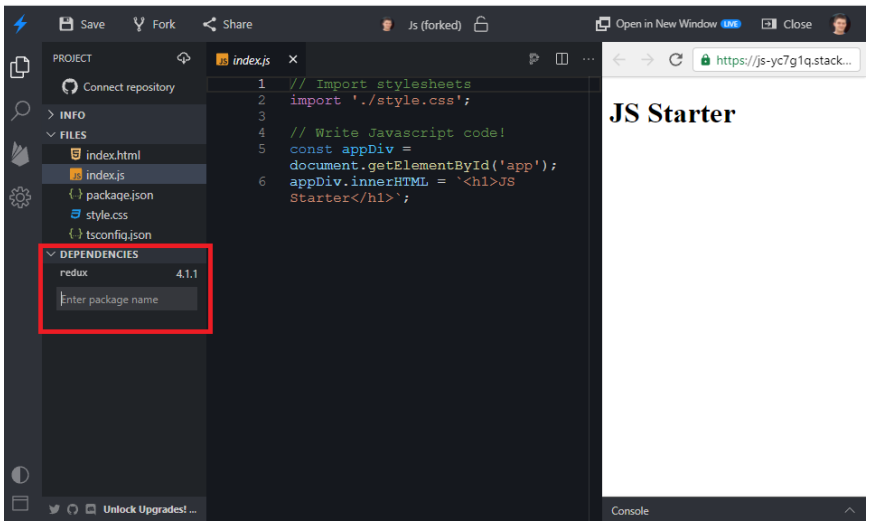
2.1 (Novo projeto Javascript) Na tela inicial do StackBlitz, clique **Javascript – Blank Project**, como destaca a Figura 2.1.1.

Figura 2.1.1



2.2 (Adicionando o Redux como dependência) O Redux é uma biblioteca Javascript e precisa ser adicionado como dependência do projeto. Para tal, clique sob "DEPENDENCIES', digite "redux" e aperte Enter. Veja a Figura 2.2.1.

Figura 2.2.1



2.3 (Um criador de ação para a criação de contratos) Uma das solicitações previstas no sistema é a criação de contratos. Isso é responsabilidade de um criador de ações que, neste caso, será uma simples função. Ela recebe nome e valor da taxa de criação de contrato e devolve uma **ação**. A ação é um simples objeto JSON contendo tipo e os dados de interesse. Veja o Bloco de Código 2.3.1. Estamos no arquivo **index.js**. Caso ele possua algum conteúdo inicial, apague tudo.

Nota: Lembre-se de clicar **Save** esporadicamente no canto superior esquerdo da tela.

Bloco de Código 2.3.1

```
//essa função é criadora de um tipo de ação
const criarContrato = (nome, taxa) => {
 //esse JSON que ela devolve é uma ação
 return {
  type: "CRIAR_CONTRATO",
  dados: {
   nome, taxa
  }
 }
}
```

2.4 (Um criador de ação para o cancelamento de contratos) O Bloco de Código mostra uma função que desempenha o papel de criadora de ações. Ela cria ações para o cancelamento de contratos.

Bloco de Código 2.4.1

```
//esta função é criadora de um tipo de ação
const cancelarContrato = (nome) => {
 //esse JSON que ela devolve é uma ação
 return {
  type: "CANCELAR_CONTRATO",
  dados: {
   nome
  }
 }
}
```

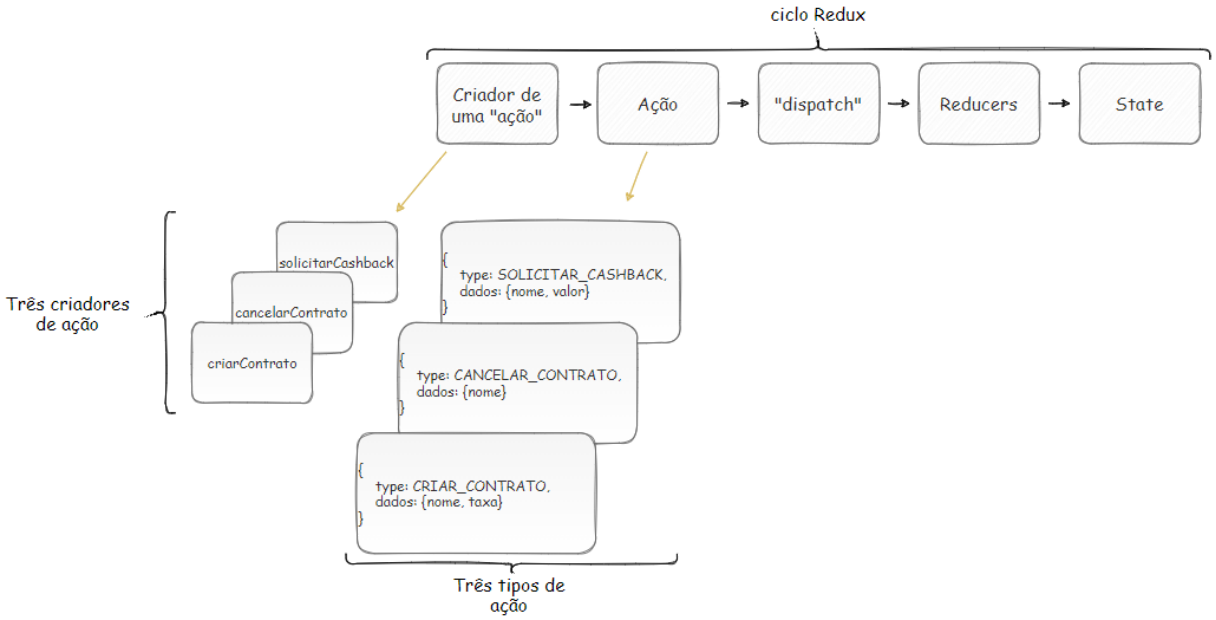
2.5 (Um criador de ações para solicitações de cashback) Por sua vez, a função exibida pelo Bloco de Código 2.5.1 cria ações para solicitações de cashback.

Bloco de Código 2.5.1

```
//esta função é criadora de um tipo de ação
const solicitarCashback = (nome, valor) => {
 //esse JSON que ela devolve é uma ação
 return {
  type: "CASHBACK",
  dados: {
   nome, valor
  }
 }
}
```

2.6 (Partes do ciclo Redux implementadas até então) A Figura 2.6.1 mostra as partes do ciclo Redux que implementamos até então.

Figura 2.6.1



2.7 (Um reducer para o tratamento de solicitações de cashback) Um reducer é uma simples função que recebe partes do estado atual que lhe sejam de interesse e a ação a ser tratada. Cabe a ela devolver o estado atualizado de acordo com os parâmetros recebidos. O reducer para o tratamento de solicitações de cashback aparece no Bloco de Código 2.7.1. Repare que ele representa o departamento da empresa responsável por essa atividade.

Bloco de Código 2.7.1

```
//esta função é um reducer
//quando chamada pela primeira vez, seu primeiro parâmetro será undefined
//já que não existirá histórico algum
//por isso, configuramos uma lista vazia como seu valor padrão
const historicoDePedidosDeCashback = (historicoDePedidosDeCashbackAtual = [], acao) => {
 //se a ação for CASHBACK, adicionamos o novo pedido à coleção existente
 if (acao.type === 'CASHBACK'){
  //uma cópia. Contém todos os existentes + o novo
  //não faça push
  return [
   ...historicoDePedidosDeCashbackAtual,
   acao.dados
  ]
 }
 //caso contrário, apenas ignoramos e devolvemos a coleção inalterada
 return historicoDePedidosDeCashbackAtual
}
```

2.8 (Um reducer para a manipulação do caixa) O reducer do Bloco de Código 2.8.1 implementa a lógica do departamento de caixa. Ele recebe o valor existente no caixa e o altera de acordo com o tipo da ação recebida.

Bloco de Código 2.8.1

```
//caixa começa zerado
const caixa = (dinheiroEmCaixa = 0, acao) => {
 if (acao.type === "CASHBACK"){
  dinheiroEmCaixa -= acao.dados.valor
 }
 else if (acao.type === "CRIAR_CONTRATO"){
  dinheiroEmCaixa += acao.dados.taxa
 }
 return dinheiroEmCaixa
}
```

2.9 (Um reducer para a criação e cancelamento de contratos) O reducer – uma simples função, lembra? – do Bloco de Código 2.9.1 trata a criação e cancelamento de contratos.

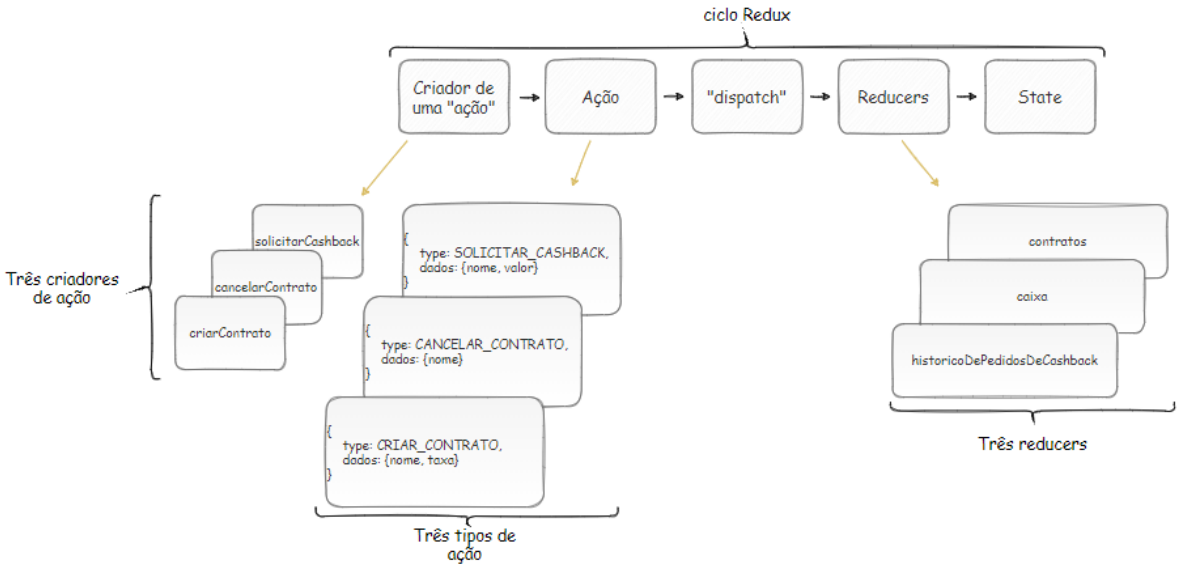
Bloco de Código 2.9.1

```
//lista começa vazia
```

```
const contratos = (listaDeContratosAtual = [], acao) => {
  if (acao.type === "CRIAR_CONTRATO")
    return [...listaDeContratosAtual, acao.dados]
  if (acao.type === "CANCELAR_CONTRATO")
    return listaDeContratosAtual.filter(c => c.nome !== acao.dados.nome)
  return listaDeContratosAtual
}
```

2.10 (Partes do ciclo Redux implementadas até então) A Figura 2.10.1 atualiza as partes do ciclo Redux que implementamos até então. Temos agora os reducers, veja.

Figura 2.10.1



2.11 (Utilizando o Redux) Até o momento sequer fizemos uso do Redux. Escrevemos apenas algumas funções Javascript comuns. Passaremos a utilizá-lo nesta seção. Quando utilizamos o Redux, combinamos coleções de criações de ações com reducers apropriados, obtendo um maquinário capaz de desempenhar o gerenciamento do estado centralizado. O primeiro passo é trazer o Redux para o contexto, como mostra o Bloco de Código 2.11.1. Você pode importá-lo na primeira linha de código do arquivo **index.js**, antes de definir qualquer função.

Bloco de Código 2.11.1

```
const Redux = require ('redux')
...
```

A seguir, após a definição de todas as funções, desestruturamos o objeto Redux a fim de obter as funções **createStore** e **combineReducers**. Veja o Bloco de Código 2.11.2.

Bloco de Código 2.11.2

```
...
//depois da definição de todas as funções
const { createStore, combineReducers } = Redux
```

O próximo passo é combinar todos os reducers utilizando a função **combineReducers**. Veja o Bloco de Código 2.11.3.

Bloco de Código 2.11.3

```
//depois da definição de todas as funções
const { createStore, combineReducers } = Redux

const todosOsReducers = combineReducers({
  historicoDePedidosDeCashback,
  caixa,
  contratos
})
```

Nota: O nome de cada "parte" do estado gerenciado pelo reducer será determinado pela chave escolhida quando **combineReducers** foi chamada. Neste caso, elas serão

historicoDePedidosDeCashback, **caixa** e **contratos**. Se desejar trocar o nome, basta escolher outro nome. Veja o exemplo do Bloco de Código 2.11.4.

Bloco de Código 2.11.4 – Apenas um exemplo

```
//depois da definição de todas as funções
const { createStore, combineReducers } = Redux

const todosOsReducers = combineReducers({
  historicoCashback: historicoDePedidosDeCashback,
  nossocaixa: caixa,
  osContratos: contratos
})
```

A seguir, construímos o chamado **store** do Redux, utilizando a função intuitivamente denominada **createStore**. Veja o Bloco de Código 2.11.5.

Bloco de Código 2.11.5

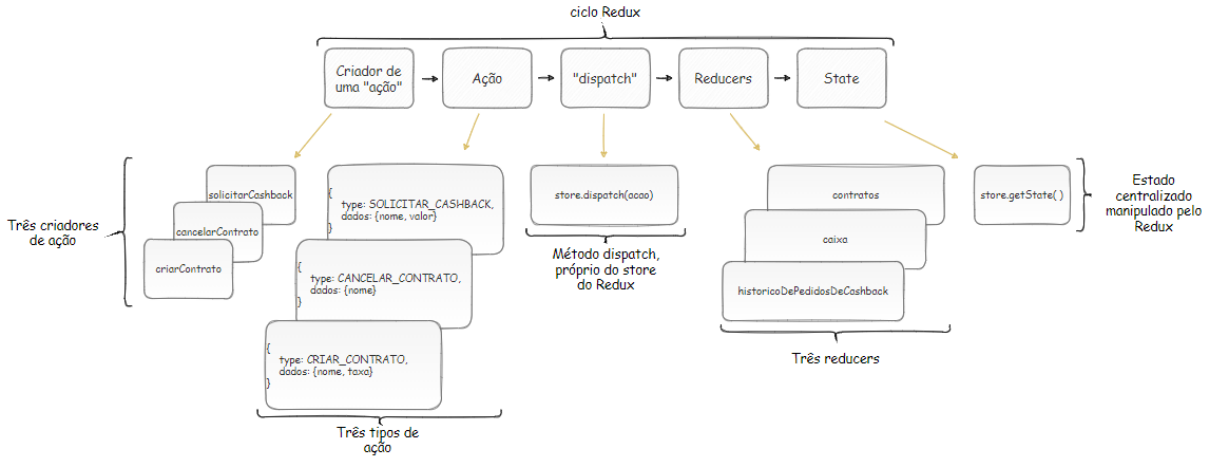
```
//depois da definição de todas as funções
const { createStore, combineReducers } = Redux

const todosOsReducers = combineReducers({
  historicoDePedidosDeCashback,
  caixa,
  contratos
})

const store = createStore(todosOsReducers)
```

2.12 (Partes do ciclo Redux implementadas até então) A Figura 2.12.1 exibe a implementação completa do ciclo Redux. Observe que o objeto **store** que criamos possui um método chamado **getState**. O objeto devolvido por ele desempenha o papel de estado. Ele pode ser usado assim: **store.getState()**. E o item "dispatch"? Ele é um método do objeto store que passaremos a utilizar a seguir. Ele pode ser usado assim: **store.dispatch(acao)**.

Figura 2.12.1



2.13 (Sequência de ações enviadas ao Redux) O programa do Bloco de Código 2.13.1 realiza as seguintes tarefas em ordem:

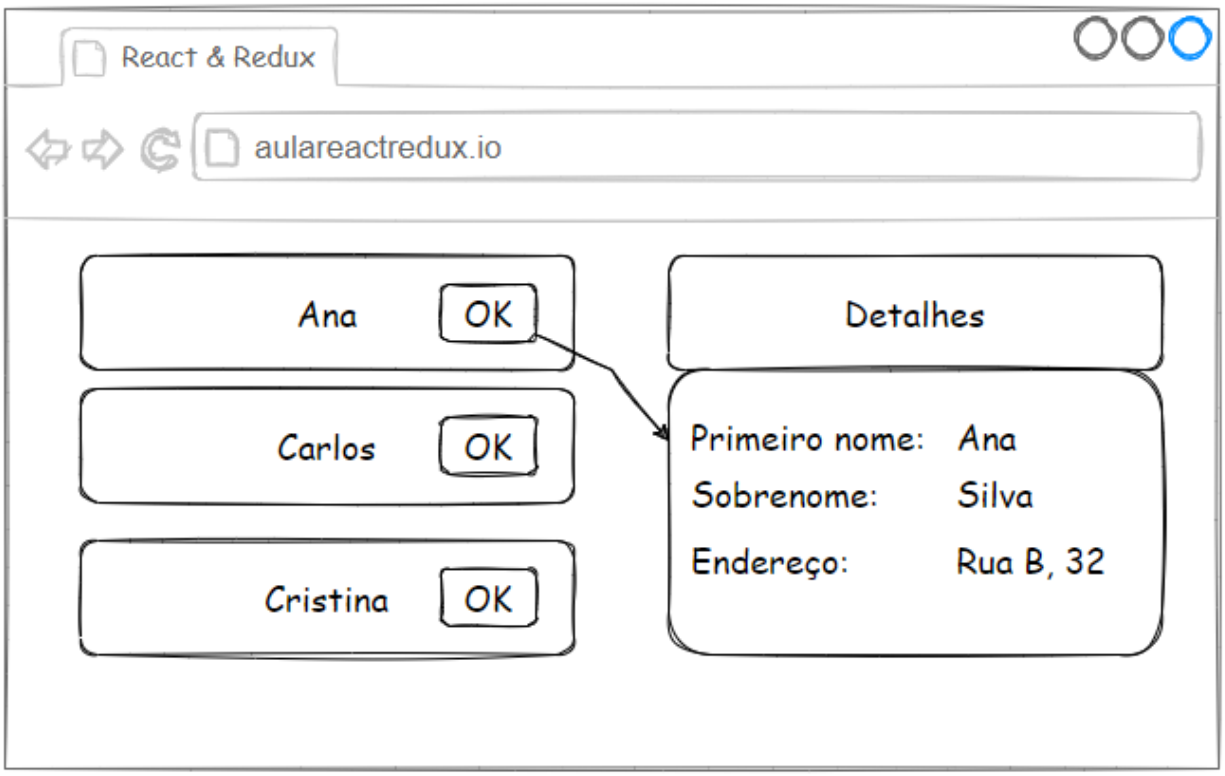
- Cria um contrato para José
- Cria um contrato para Maria
- Solicita cashback de 10 para Maria
- Solicita cashback de 20 para José
- Cancela o contrato de Maria

Observe que a execução acontece automaticamente assim que o projeto é salvo. Ele pode ser salvo automaticamente ou você pode clicar **Save** no canto superior esquerdo. Abra o console do navegador (CTRL + SHIFT + I no Chrome, por exemplo) para visualizar o resultado. Ele deve ser parecido com aquele que a Figura 2.13.1 mostra.

3 Uma aplicação React com Redux

Como vimos, o Redux é uma biblioteca de manipulação de estado que opera de maneira completamente independente. Nesta seção, veremos como utilizá-lo especificamente em uma aplicação React. A aplicação exibirá uma lista de nomes de pessoas. Quando uma delas for clicada, a aplicação exibirá mais detalhes a seu respeito. Veja a Figura 3.1.

Figura 3.1



3.1 (Criando a aplicação) As seções 3.1.1 e 3.1.2 mostram duas formas diferentes para a criação da aplicação. Use a primeira caso nunca tenha feito a instalação da PrimeReact e da PrimeFlex. São as bibliotecas de componentes e de utilitários que utilizaremos. Vale a pena fazer uma primeira vez para aprender. Caso já saiba sobre o assunto e já tenha feito anteriormente, use a segunda.

3.1.1 (Possibilidade 1: Novo projeto e componente principal. Instalação passo a passo da PrimeReact e da PrimeFlex) Crie um projeto com

npx create-react-app react-redux-pessoas

Use

cd react-redux-pessoas

para navegar até o diretório em que se encontra o seu projeto. Use

code .

para obter uma instância do VS Code vinculada a esse diretório. No VS Code, clique **Terminal >> New Terminal** para obter um novo terminal interno do VS Code, o que simplifica o trabalho. Neste terminal, digite

npm start

para colocar a aplicação em funcionamento. Uma janela do seu navegador padrão deve ser aberta fazendo uma requisição a **localhost:3000**.

Apague todos os arquivos existentes na pasta **src**. A seguir, crie uma pasta chamada **components**, subpasta de **src**. Definiremos todos os componentes dessa aplicação em arquivos dentro dela.

Na pasta **components**, crie um arquivo chamado **App.js**. Veja seu conteúdo inicial no Bloco de Código 3.1.1.1.

Bloco de Código 3.1.1.1

```
import React from 'react'
const App = () => {
    return (
        <div>
            App
        </div>
    )
}
export default App
```

(Dependências) Utilizaremos componentes da biblioteca PrimeReact e os utilitários e grid system da PrimeFlex. As suas respectivas documentações podem ser visitadas a seguir.

PrimeReact
https://www.primefaces.org/primereact/

PrimeFlex
https://www.primefaces.org/primeflex/

Faça a instalação da PrimeReact com

npm install primereact
npm install primeicons
npm install react-transition-group

A PrimeFlex pode ser instalada com

npm install primeflex

A seguir, importe o CSS da PrimeReact, PrimeIcons, PrimeFlex e um dos temas descritos na documentação, na página **Get Started**. Isso pode ser feito no arquivo **index.js**, que deve ser criado na pasta **src**. Veja o Bloco de Código 3.1.1.2.

Bloco de Código 3.1.1.2

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import 'primereact/resources/primereact.min.css'
import 'primeicons/primeicons.css'
import 'primeflex/primeflex.css'
import 'primereact/resources/themes/bootstrap4-light-purple/theme.css'
ReactDOM.render(
    <App />,
    document.querySelector('#root')
)
```

3.1.2 (Possibilidade 2: Obtendo um modelo inicial que já contém as dependências instaladas) Caso não tenha realizado os passos da seção 3.1.1, você pode obter uma cópia do projeto com as dependências configuradas. Há um modelo disponível aqui.

https://github.com/professorbossini/pessoal_react_modelo_primereact_primeflex

Para obter uma cópia para você, execute o seguinte comando usando um terminal vinculado ao diretório que representa o seu workspace (fora de qualquer outro projeto React!)

git clone
https://github.com/professorbossini/pessoal_react_modelo_primereact_primeflex
react-redux-pessoas

O projeto obtido conterá um arquivo chamado **package.json** que descreve as dependências. Para fazer o seu download, navegue até a pasta do projeto com

cd react-redux-pessoas

A seguir, use o seguinte comando para instalar as dependências.

npm install

Neste momento já deve ser possível colocar o projeto em funcionamento com o seguinte comando.

npm start

Deve ser possível acessar a aplicação no endereço a seguir a partir de agora.

localhost:3000

A seguir, pode ser interessante alterar a URL associada ao remote **origin** utilizando uma URL de um repositório próprio seu. Assim você poderá fazer seu próprio controle de versão. Para tal, depois de criar um repositório no Github, use

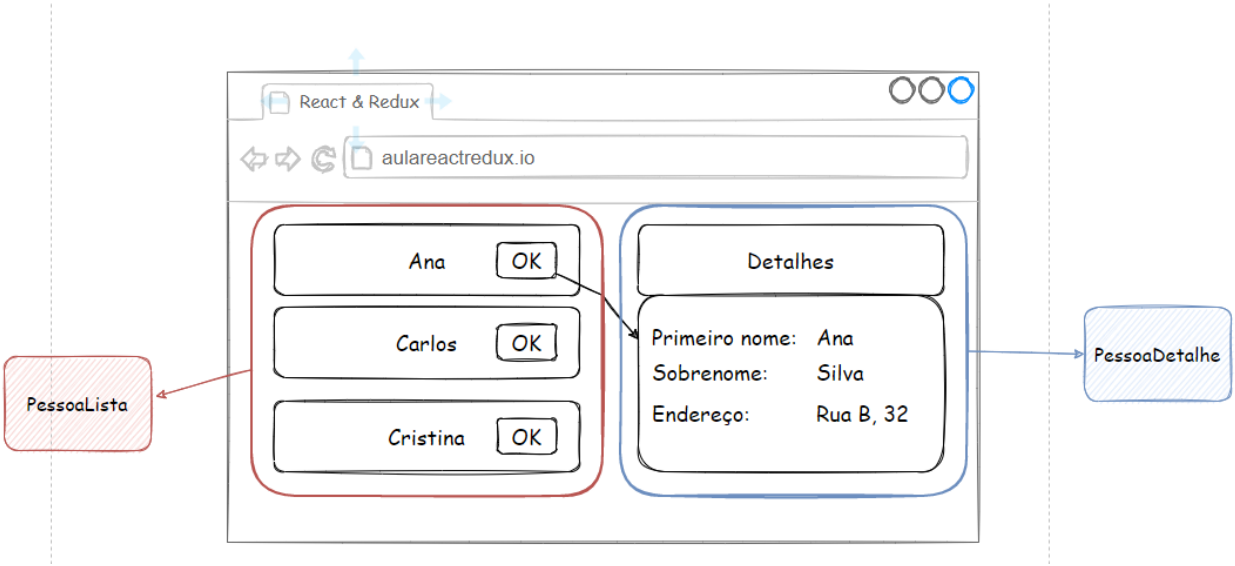
git remote set-url origin url-do-seu-repositorio-no-github

A partir de agora, as operações push envolvendo o remote origin serão direcionadas ao seu repositório remoto. Aliás, você já pode executar o seguinte comando para ter certeza de que tudo está configurado corretamente.

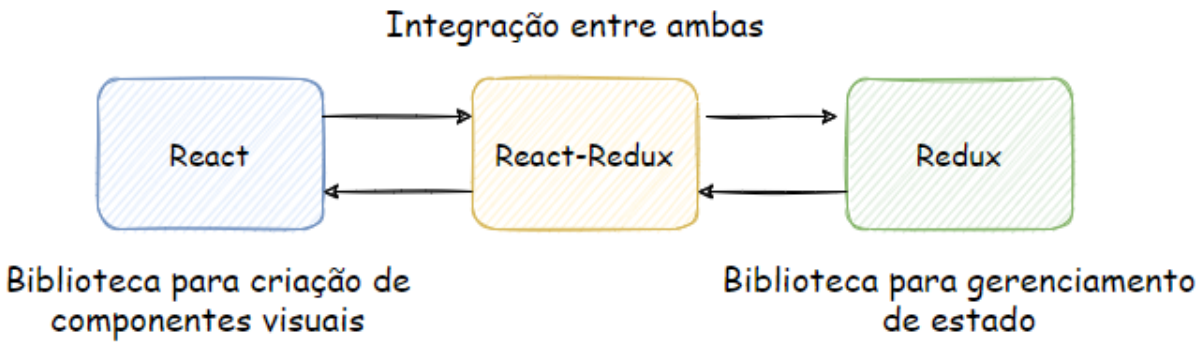
git push -u origin master

3.2 (Os componentes da aplicação) Nossa aplicação terá apenas dois componentes React. Veja a Figura 3.2.1.

Figura 3.2.1



3.3 (As bibliotecas) Nas últimas aulas, já utilizamos diversos recursos do React. Quanto ao Redux, aprendemos a utilizar alguns de seus recursos e vimos que ele é independente de qualquer biblioteca, inclusive do próprio React. É muito comum, entretanto, escrever aplicações que fazem uso de ambas, como é o caso desta aplicação que estamos escrevendo agora. A integração entre ambas se dá por meio de uma outra biblioteca chamada **react-redux**. Veja a Figura 3.3.1.



Visite a documentação da biblioteca react-redux por meio do Link 3.3.1.

Link 3.3.1
https://react-redux.js.org/

Use

npm install redux

para fazer a instalação do Redux. A seguir, use

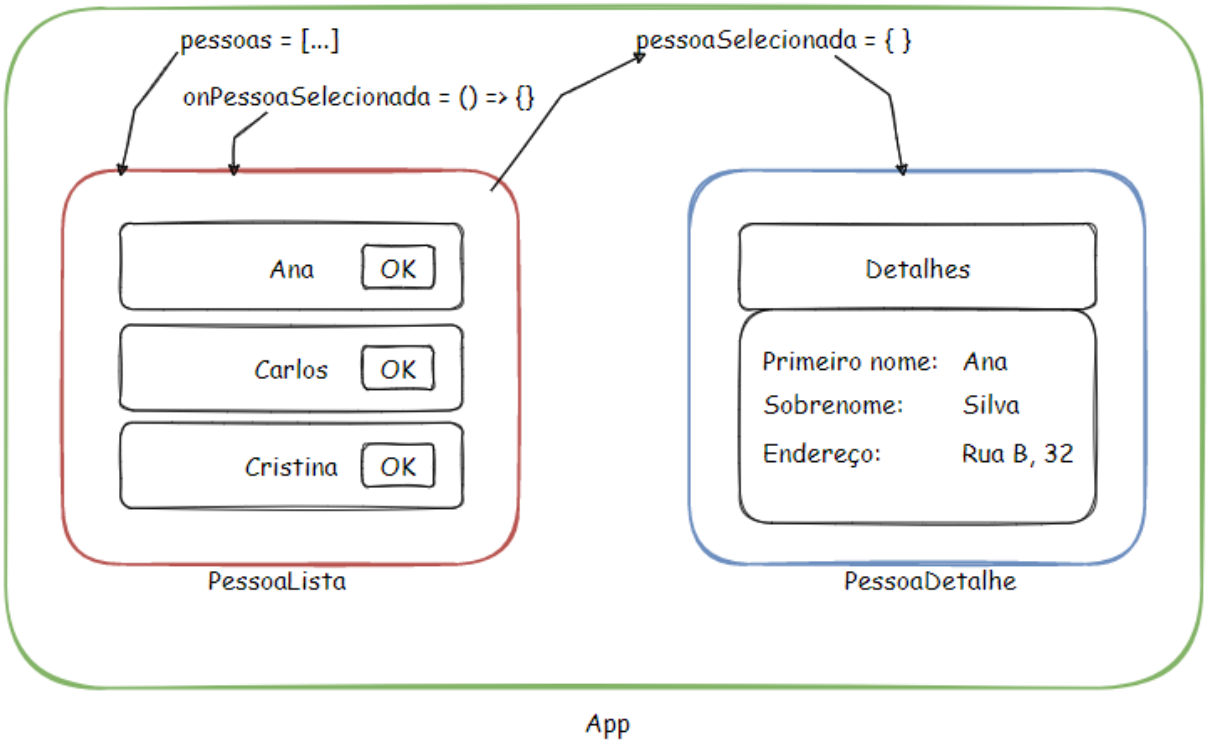
npm install react-redux

para fazer a instalação da biblioteca de integração. Ela traz diversas funções utilitárias que simplificarão a integração entre React e Redux.

3.4 (Abordagens sem e com Redux) A Figura 3.4.1 mostra uma possível abordagem de implementação para esta aplicação sem utilizar o Redux. Os principais pontos são os seguintes.

I. O componente principal armazena uma lista de pessoas e a pessoa selecionada.
II. O componente principal envia para o componente PessoaLista, via props, a lista de pessoas e uma função a ser chamada quando uma pessoa for selecionada.
III. O componente PessoaLista chama a função recebida alterando o estado do componente principal.
IV. O componente principal envia para o componente PessoaDetalhe, via props, a pessoa selecionada para que ela possa ser exibida.

Figura 3.4.1

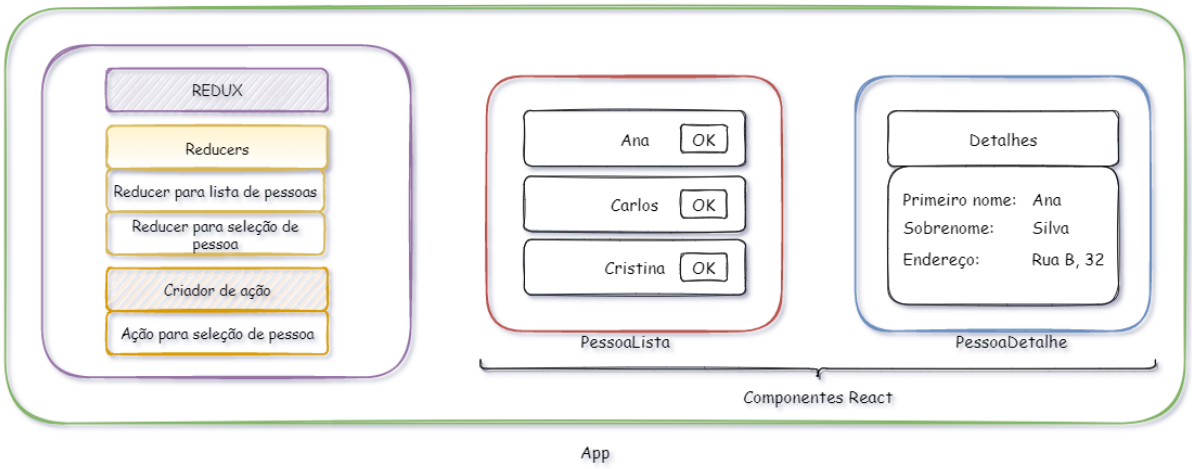


Utilizando o Redux, podemos definir reducers para

- criação da lista de pessoas
- seleção de pessoa

E um criador de ação utilizado quando uma pessoa for selecionada. Veja a Figura 3.4.2.

Figura 3.4.2



3.5 (Pasta e arquivo index.js para o criador de ações: o criador de ações) Crie uma pasta chamada **actions**, subpasta de **src**. Nela, crie um arquivo chamado **index.js**. Utilizando este nome, podemos importar o conteúdo especificando somente a pasta em que ele reside, não sendo necessário especificar o nome do arquivo. O Bloco de Código 3.5.1 mostra a definição do único criador de ações da aplicação.

Bloco de Código 3.5.1

```
//essa função cria uma ação
export const selecionarPessoa = (pessoa) => {
    //esse JSON é a ação criada
    return {
        type: "PESSOA_SELECIONADA",
        dados: pessoa
    }
}
```

3.6 (Os reducers) Crie uma pasta chamada **reducers** e, dentro dela, um arquivo chamado **index.js**. O primeiro reducer que definiremos se encarrega de produzir a lista de pessoas. Observe que ele é bastante simples e não requer nenhum parâmetro. Ele apenas devolve uma lista estática de pessoas. Veja o Bloco de Código 3.6.1.

Bloco de Código 3.6.1

```
const pessoasReducer = () => {
    return [
        {
            nome: 'Cristina', sobrenome: "Silva", endereco: "Rua A, 34"
        },
        {
            nome: "João", sobrenome: "Alves", endereco: "Rua B, 43"
        },
        {
            nome: "Pedro", sobrenome: "Mendes", endereco: "Rua D, 200"
        }
    ]
}
```

A seguir, definimos o reducer para "pessoa selecionada". Ele recebe a pessoa selecionada e uma ação. Se a ação for do tipo adequado, apenas devolve seu objeto "dados", já que ele armazena a pessoa selecionada. Caso contrário, devolve a pessoa previamente selecionada. Repare na sua simplicidade. Veja o Bloco de Código 3.6.2.

Bloco de Código 3.6.2

```
const pessoasReducer = () => {
    return [
        {
            nome: 'Cristina', sobrenome: "Silva", endereco: "Rua A, 34"
        },
        {
            nome: "João", sobrenome: "Alves", endereco: "Rua B, 43"
        },
        {
            nome: "Pedro", sobrenome: "Mendes", endereco: "Rua D, 200"
        }
    ]
}

const pessoaSelecionadaReducer = (pessoaSelecionada = null, acao) => {
    if (acao.type === 'PESSOA_SELECIONADA'){
        return acao.dados
    }
    return pessoaSelecionada
}
```

Por fim, **combinamos** os reducers e exportamos o objeto resultante. Veja o Bloco de Código 3.6.3.

Bloco de Código 3.6.3

```
import { combineReducers } from "redux"
const pessoasReducer = () => {
    return [
        {
            nome: 'Cristina', sobrenome: "Silva", endereco: "Rua A, 34"
        },
        {
            nome: "João", sobrenome: "Alves", endereco: "Rua B, 43"
        },
        {
            nome: "Pedro", sobrenome: "Mendes", endereco: "Rua D, 200"
        }
    ]
}

const pessoaSelecionadaReducer = (pessoaSelecionada = null, acao) => {
    if (acao.type == 'PESSOA_SELECIONADA'){
        return acao.dados
    }
    return pessoaSelecionada
}

export default combineReducers({
    pessoas: pessoasReducer,
    pessoaSelecionada: pessoaSelecionadaReducer
})
```

3.7 (O Provider) Provider é um componente disponibilizado pela **react-redux**. Nossos componentes React precisam de acesso ao objeto **store** do Redux. Podemos entregá-lo aos componentes utilizando um Provider. Basta que uma tag Provider englobe os componentes React. Veja o Bloco de Código 3.7.1. Estamos no arquivo **src/index.js**.

Bloco de Código 3.7.1

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './components/App'
import 'primereact/resources/primereact.min.css'
import 'primeicons/primeicons.css'
import 'primeflex/primeflex.css'
// escolha o tema que desejar
// https://primefaces.org/primereact/showcase/#/setup
//escolhemos esse aqui
import 'primereact/resources/themes/bootstrap4-light-purple/theme.css'

import { Provider } from 'react-redux'
import  { createStore } from 'redux'
import reducers from  './reducers'

ReactDOM.render(
    <Provider store={createStore(reducers)}>
        <App/>
    </Provider>,
    document.querySelector('#root')
)
```

3.8 (O componente PessoaLista) Crie um arquivo chamado PessoaLista.js na pasta **components** para definir o componente PessoaLista. Veja sua definição inicial no Bloco de Código 3.8.1.

Bloco de Código 3.8.1

```
import React, { Component } from 'react'
class PessoaLista extends Component{
    render(){
        return <div>
                 Pessoas
        </div>
    }
}
export default PessoaLista
```

No componente App (arquivo **App.js**), utilize o componente PessoaLista como no Bloco de Código 3.8.2.

Bloco de Código 3.8.2

```
import React from 'react'
import PessoaLista from './PessoaLista'
const App = () => {
    return (
        <div>
            <PessoaLista />
        </div>
    )
}
export default App
```

Para que o componente PessoaLista tenha acesso aos recursos do Redux, utilizamos a função **connect**. Quando chamada, ela devolve uma função que, então, colocamos em execução entregando-lhe como parâmetro o nome do componente. Veja o Bloco de Código 3.8.4. Estamos no arquivo **PessoaLista.js**.

Bloco de Código 3.8.4

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
class PessoaLista extends Component{
    render(){
        return <div>
            Pessoas
        </div>
    }
}
export default connect()(PessoaLista)
```

3.9 (A função mapStateToProps) A função mapStateToProps tem como finalidade fazer com que o estado gerenciado pelo Redux seja entregue aos componentes via props. O nome faz sentido. Entretanto, trata-se de uma convenção e seu nome pode ser qualquer um. Conheça mais sobre o funcionamento de **mapStateToProps** no Link 3.8.1.

Link 3.8.1
https://react-redux.js.org/using-react-redux/connect-mapstate

Veja uma definição inicial no Bloco de Código 3.9.1. Ali, apenas exibimos o estado para entender o que está acontecendo. Estamos no arquivo **PessoaLista.js**.

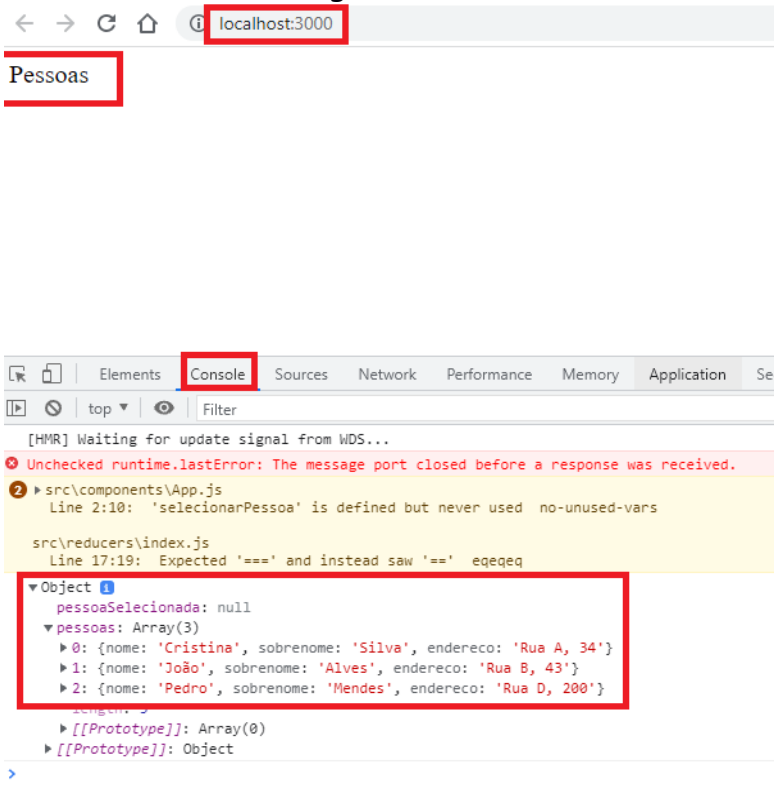
Bloco de Código 3.9.1

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
class PessoaLista extends Component{
    render(){
        console.log (this.props)
        return <div>
            Pessoas
        </div>
    }
}

//esse nome é uma convenção apenas
const mapStateToProps = (state) => {
    return {pessoas: state.pessoas }
}
//A função connect é chamada
//Ela recebe mapStateToProps como parâmetro
//a expressão connect(mapStateToProps) resulta em uma função
//ela é chamada com PessoaLista como parâmetro
//O props de PessoaList passa a ter acesso ao estado
export default connect(mapStateToProps)(PessoaLista)
```

Neste momento, visite **localhost:3000** e abra o console do seu navegador. Você deverá ver um resultado parecido com aquele que a Figura 3.9.1 exibe.

Figura 3.9.1



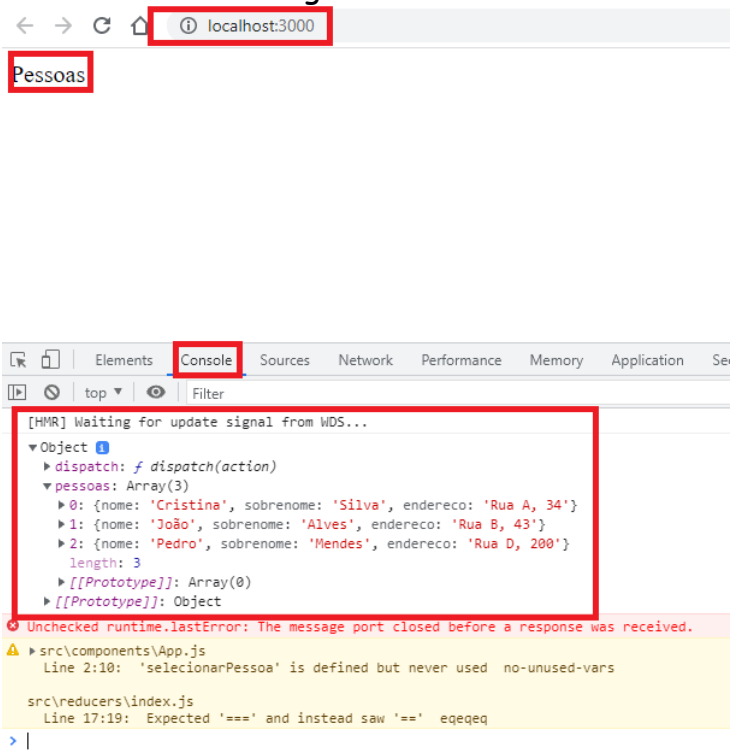
Observe que o estado possui tanto a pessoa selecionada quanto a lista de pessoas. O componente PessoaLista está interessado apenas na lista de pessoas. Por isso, faremos com que a função devolva somente esta parte de interesse. Observe que exibimos o objeto props do componente no console do navegador. Veja o Bloco de Código 3.9.3.

Bloco de Código 3.9.3

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
class PessoaLista extends Component{
    render(){
        console.log (this.props)
        return <div>
            Pessoas
        </div>
    }
}
const mapStateToProps = (state) => {
    return {pessoas: state.pessoas }
}
export default connect(mapStateToProps)(PessoaLista)
```

Visite novamente localhost:3000 e verifique o resultado no console. Ele deve ser parecido com aquele que a Figura 3.9.2 exibe.

Figura 3.9.2



Talvez seja mais didático armazenar a função que a connect produz e, num segundo passo, colocá-la em execução. Veja como fazê-lo no Bloco de Código 3.9.3.

Bloco de Código 3.9.3

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
class PessoaLista extends Component{
    render(){
        console.log (this.props)
        return <div>
            Pessoas
        </div>
    }
}

//esse nome é uma convenção apenas
const mapStateToProps = (state) => {
    console.log(state)
    return {pessoas: state.pessoas }
}
//A função connect é chamada
//Ela recebe mapStateToProps como parâmetro
//a expressão connect(mapStateToProps) resulta em uma função
//ela é chamada com PessoaLista como parâmetro
//O props de PessoaLista passa a ter acesso ao estado
// export default connect(mapStateToProps)(PessoaLista)
// talvez mais didático
const resultadoDaConnect = connect(mapStateToProps)
//esta função tem acesso ao estado e ao componente
//assim, ela pode torná-lo disponível via props do componente
export default resultadoDaConnect (PessoaLista)
```

3.10 (Exibindo a lista) A exibição da lista pode ser feita com uma expressão JSX simples. Lembre-se de utilizar os recursos da PrimeReact e da PrimeFlex. Repare que estamos usando o nome como chave. Para este momento, vamos considerar suficiente. Veja uma possível definição no Bloco de código 3.10.1. Estamos no arquivo **PessoaLista.js**.

Bloco de Código 3.10.1

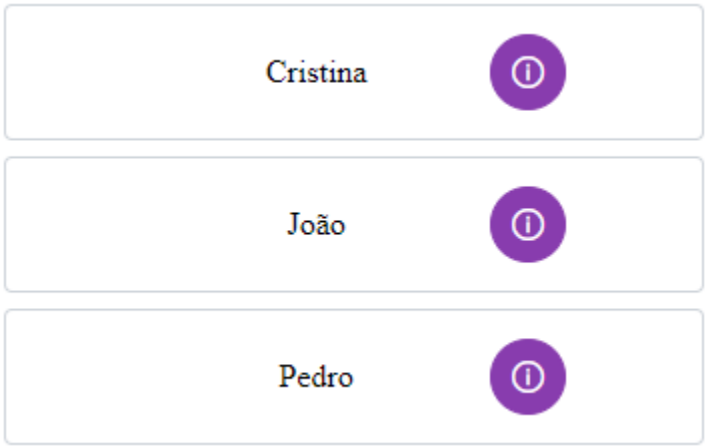
```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { Button } from 'primereact/button'
class PessoaLista extends Component{
    render(){
        return (
            this.props.pessoas.map((pessoa => (
                //filhos em linha, margem abaixo de cada um, borda, cen-
tralizado horizontalmente
                <div key={pessoa.nome} className="flex flex-row mb-2 w-6
                    border border-round border-1 border-400
                                justify-content-center">
                    {/* padding, largura */}
                    <div className="p-2 w-6">
                        <p className="text-center">
                           {pessoa.nome}
                        </p>
                    </div>
                    {/* flex e centralizando nos dois eixos */}
                    <div className="flex flex-row justify-content-center
                       align-items-center">
                        <Button
                            icon="pi pi-info-circle"
                            className="p-button-rounded"
                        />
                    </div>
                </div>
            )))
        )
    }
}

//esse nome é uma convenção apenas
const mapStateToProps = (state) => {
    console.log(state)
    return {pessoas: state.pessoas }
}

const resultadoDaConnect = connect(mapStateToProps)
export default resultadoDaConnect (PessoaLista)
```

O resultado deve ser parecido com o que exibe a Figura 3.10.1. Como não estamos fazendo uso de nenhum mecanismo para ajuste de layout, como o próprio grid system, é natural que a exibição ocorra a partir do canto superior esquerdo, sem centralização.

Figura 3.10.1



3.11 (Selecionando uma pessoa da lista) Cada item na lista tem um botão associado. Quando clicado, um botão deve fazer com que os detalhes da pessoa associada sejam exibidos. Clicar em um botão, portanto, significa "selecionar uma pessoa". Isso será feito por meio da **criação de uma ação do tipo PESSOA_SELECIONADA**. Lembre-se que nosso criador de ações está definido no arquivo **src/actions/index.js**. Por ser definido em um arquivo chamado **index.js**, ele pode ser importado sem que o nome do arquivo seja especificado. O componente PessoaLista precisa de acesso a ele para que possa fazer o vínculo com os botões. Tal como feito com a lista de pessoas, o criador de ações será disponibilizado a ele via props, por meio do uso da função **connect**. Veja o Bloco de Código 3.11.1. Estamos no arquivo **PessoaLista.js**.

Bloco de Código 3.11.1

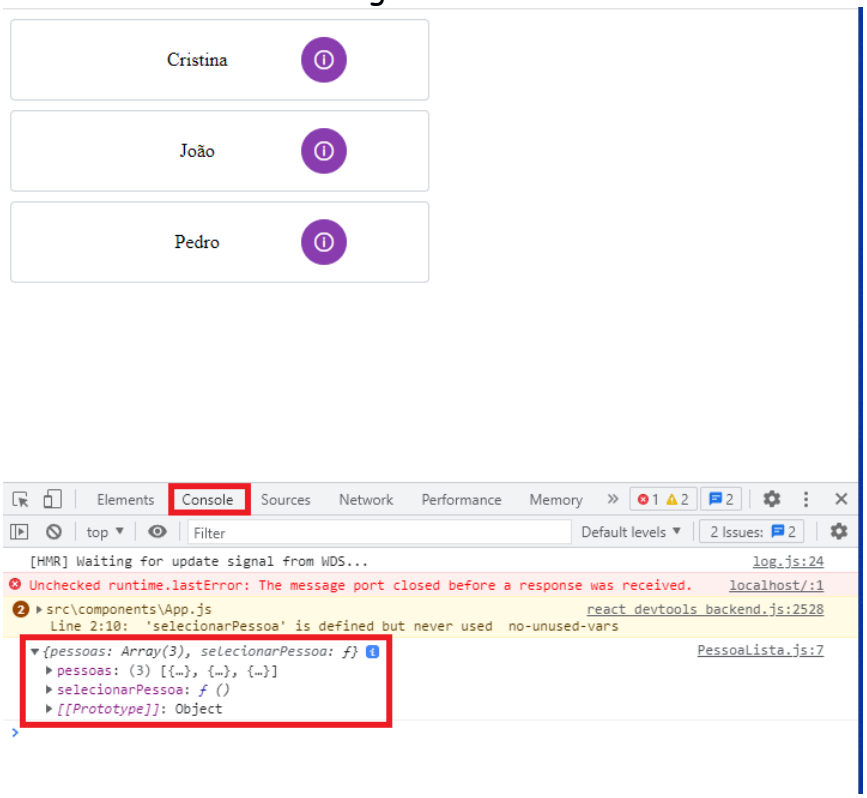
```
...
import { selecionarPessoa } from '../actions'
class PessoaLista extends Component{
    render(){
        console.log(this.props)
        return (
            this.props.pessoas.map((pessoa => (
                //filhos em linha, margem abaixo de cada um, borda, cen-
tralizado horizontalmente
                <div key={pessoa.nome} className="flex flex-row mb-2 w-6
border border-round border-1 border-400 justify-content-center">
                    {/* padding, largura */}
                    <div className="p-2 w-6">
                        <p className="text-center">{pessoa.nome}</p>
                    </div>
                    {/* flex e centralizando nos dois eixos */}
                    <div className="flex flex-row justify-content-center
align-items-center">
                        <Button
                            icon="pi pi-info-circle"
                            className="p-button-rounded"
                        />
                    </div>
                </div>
            )))
        )
    }
}
//esse nome é uma convenção apenas
const mapStateToProps = (state) => {
    return {pessoas: state.pessoas }
}

const resultadoDaConnect = connect(
    mapStateToProps,
    {selecionarPessoa}
)

export default resultadoDaConnect (PessoaLista)
```

A instrução console.log destacada deve fazer com que um objeto parecido com aquele exibido pela Figura 3.11.1 seja exibido no console do navegador. Repare que a função **selecionarPessoa** agora faz parte do objeto props do componente.

Figura 3.11.1



A função selecionarPessoa pode ser vinculada ao botão como destaca o Bloco de Código 3.11.2. Repare que adicionamos uma instrução **console.log** ao corpo da função **mapStateToProps**. Ocorre que ela é chamada sempre que o estado é atualizado, o que é feito sempre que um botão é clicado. Assim, poderemos ver a pessoa atualmente selecionada sempre que clicarmos num botão.

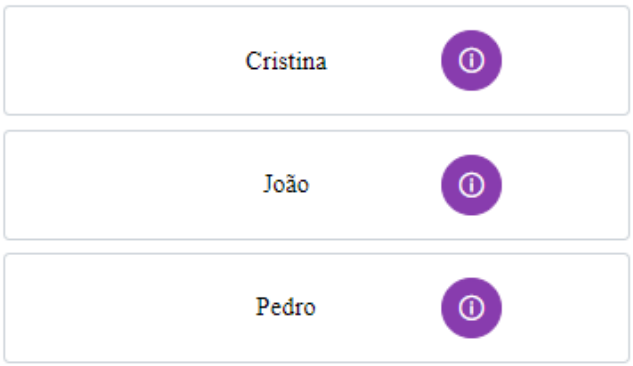
Bloco de Código 3.11.2

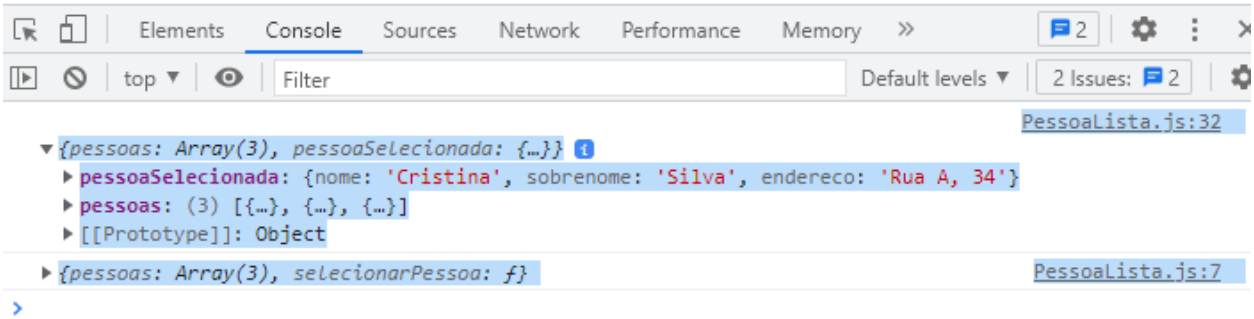
```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { Button } from 'primereact/button'
import { selecionarPessoa } from '../actions'
class PessoaLista extends Component{
    render(){
        console.log(this.props)
        return (
            this.props.pessoas.map((pessoa => (
                //filhos em linha, margem abaixo de cada um, borda, centralizado
horizontalmente
                <div key={pessoa.nome} className="flex flex-row mb-2
                 border border-round border-1 border-400 justify-content-center">
                    {/* padding, largura */}
                    <div className="p-2 w-6">
                        <p className="text-center">{pessoa.nome}</p>
                    </div>
                    {/* flex e centralizando nos dois eixos */}
                    <div className="flex flex-row justify-content-center align-
items-center">
                        <Button
                            icon="pi pi-info-circle"
                            className="p-button-rounded"
                            onClick={() => this.props.selecionarPessoa(pessoa)}
                        />
                    </div>
                </div>
            )))
        )
    }
}
//esse nome é uma convenção apenas
const mapStateToProps = (state) => {
    console.log(state)
    return { pessoas: state.pessoas }
}
const resultadoDaConnect = connect(
    mapStateToProps,
    {selecionarPessoa}
)

export default resultadoDaConnect (PessoaLista)
```

Visite localhost:3000 no seu navegador e clique em algum botão. No Chrome Dev Tools (CTRL + SHIFT + I), verifique se o resultado é parecido com o que exibe a Figura 3.11.2.

Figura 3.11.2





3.12 (O componente PessoaDetalhe) É chegada a hora de implementar o componente PessoaDetalhe. Lembre-se de que ele será exibido de acordo com as seleções que o usuário fizer no componente PessoaLista: um botão clicado faz com que os dados da pessoa associada sejam exibidos por ele. Para defini-lo, crie um arquivo chamado **PessoaDetalhe.js** na pasta **components**. Veja o seu conteúdo inicial no Bloco de Código 3.12.1. Observe que criamos um componente usando uma função. Assim podemos estudar a forma como a função **connect** pode ser usada com componentes definidos dessa forma.

Bloco de Código 3.12.1

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

const PessoaDetalhe = () => {
    return <div>
        PessoaDetalhe
    </div>
}

export default PessoaDetalhe
```

A seguir, definimos a função – comumente, mas não obrigatoriamente – chamada **mapStateToProps**. Ela recebe o estado inteiro e devolve as partes de interesse para esse componente. Elas serão disponibilizadas a ele por meio de seu objeto props. Veja o Bloco de Código 3.12.2.

Bloco de Código 3.12.2

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

const PessoaDetalhe = () => {
    return <div>
        PessoaDetalhe
    </div>
}

const mapStateToProps = (state) => {
    return {
        // o componente poderá acessar o objeto referenciado por pessoa
        // usando seu objeto props
        pessoa: state.pessoaSelecionada
    }
}
export default PessoaDetalhe
```

Como fizemos anteriormente, chamamos a função **connect** entregando a ela a função **mapStateToProps**. Ela produz uma função que é chamada logo a seguir, recebendo o componente como parâmetro. Veja o Bloco de Código 3.12.3.

Bloco de Código 3.12.3

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

const PessoaDetalhe = () => {
    return <div>
        PessoaDetalhe
    </div>
}

const mapStateToProps = (state) => {
    return {
        // o componente poderá acessar o objeto referenciado por pessoa
        // usando seu objeto props
        pessoa: state.pessoaSelecionada
    }
}

export default connect(mapStateToProps)(PessoaDetalhe)
```

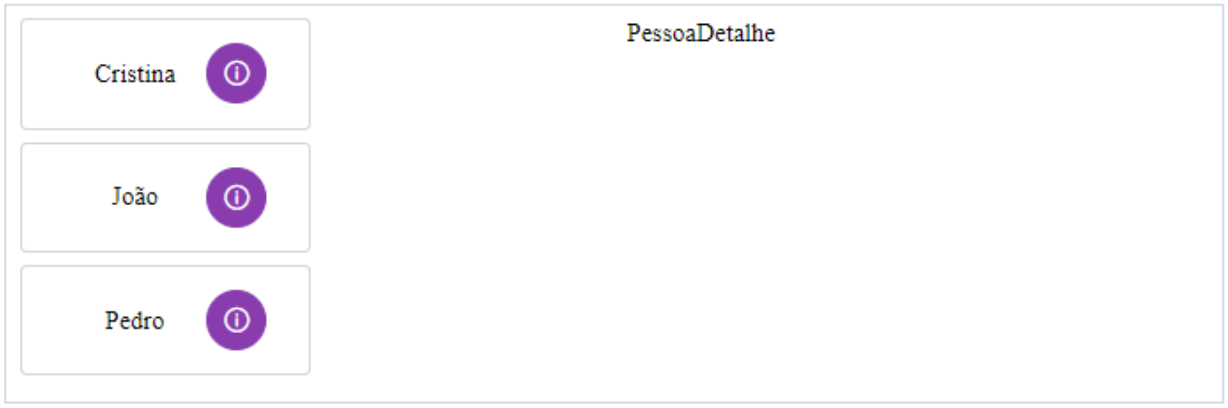
O componente será filho do componente **App**, definido no arquivo **App.js**. Veja o Bloco de Código 3.12.4.

Bloco de Código 3.12.4

```
import React from 'react'
import PessoaLista from './PessoaLista'
import PessoaDetalhe from './PessoaDetalhe'
const App = () => {
    return (
        <div className="grid border border-1 border-400 m-2">
            <div className="col-6">
                <PessoaLista />
            </div>
            <div className="col-6">
                <PessoaDetalhe />
            </div>
        </div>
    )
}
export default App
```

O resultado esperado aparece na Figura 3.12.1.

Figura 3.12.1



No arquivo **PessoaDetalhe.js**, faça o ajuste destacado no Bloco de Código 3.12.5 e faça novos testes clicando nos botões no navegador. Você deverá ver o nome da pessoa selecionada.

Bloco de Código 3.12.5

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

const PessoaDetalhe = (props) => {
    return <div>
        {props.pessoa?.nome}
    </div>
}

const mapStateToProps = (state) => {
    return {
        // o componente poderá acessar o objeto referenciado por pessoa
        // usando seu objeto props
        pessoa: state.pessoaSelecionada
    }
}


export default connect(mapStateToProps)(PessoaDetalhe)
```

Observe a notação **props.pessoa?.nome**. A notação ?. caracteriza o uso do operador de **encadeamento opcional**. Estamos tomando o cuido de somente acessar a propriedade **nome** de **pessoa.nome** caso, de fato, exista uma pessoa selecionada. Assim que a aplicação inicia, não há pessoa alguma selecionada e, portanto, o acesso sem verificação causaria um erro em tempo de execução. Veja mais sobre o operador de encadeamento opcional no Link 3.12.1.

Link 3.12.1
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining

Resta fazer a exibição dos demais detalhes da pessoa selecionada. Observe, no Bloco de Código 3.12.6, como optamos por fazê-lo usando um Card da PrimeReact. Estamos no arquivo **PessoaDetalhe.js**.

Bloco de Código 3.12.6

```
import React from 'react'
import { connect } from 'react-redux'
import { Card } from 'primereact/card'
const PessoaDetalhe = (props) => {
    return <Card title="Detalhes">
        <h3 className="text-center">{props.pessoa?.nome}
{props.pessoa?.sobrenome}</h3>
        <p className="text-center">{props.pessoa?.endereco}</p>
    </Card>
}

const mapStateToProps = (state) => {
    return {
        // o componente poderá acessar o objeto referenciado por pessoa
        // usando seu objeto props
        pessoa: state.pessoaSelecionada
    }
}


export default connect(mapStateToProps)(PessoaDetalhe)
```

Referências

React – A JavaScript library for building user interfaces. 2021. Disponível em <https://reactjs.org/>. Acesso em agosto de 2021.

Redux - A predictable state container for JavaScript apps. | Redux. 2021. Disponível em <https://redux.js.org>. Acesso em outubro de 2021.