ReactJS
Definição de componentes usando classes

1 Introdução

Componentes React podem ser definidos de duas formas:

- Utilizando funções (comuns ou arrow functions)
- Utilizando classes

Independentemente da forma como um componente React é definido, o seu funcionamento é sempre o mesmo. Ele deve produzir HTML (mais comumente utilizando JSX) que descreve seu aspecto visual e lidar com eventos gerados por interações com o usuário.

Versões do React anteriores à versão **16.8** tinham as características descritas na Tabela 1.1.

Tabela 1.1 - Versões do React anteriores à versão 16.8

Funcionalidade	**Componentes definidos utilizando classes**	**Componentes funcionais**
Produção de JSX	Sim, o método render se encarrega desta tarefa.	Sim, a própria função que define o componente se encarrega desta tarefa.
Execução de código em momentos específicos, como quando o componente é inserido na árvore DOM.	Sim, usando os métodos do ciclo de vida.	Não
Uso do mecanismo de "estado"	Sim, um objeto JSON representa o estado de cada componente.	Não

Desde a versão 16.8 do React, graças ao mecanismo conhecido como **Hooks**, componentes funcionais também podem executar código em momentos específicos e podem possuir "estado". Veja a Tabela 1.2.

Tabela 1.2 – Versões do React a partir da versão 16.8

Funcionalidade	**Componentes definidos utilizando classes**	**Componentes funcionais**
Produção de JSX	Sim, o método render se encarrega desta tarefa.	Sim, a própria função que define o componente se encarrega desta tarefa.
Execução de código em momentos específicos, como quando o componente é inserido na árvore DOM.	Sim, usando os métodos do ciclo de vida.	Sim, utilizando Hooks.
Uso do mecanismo de "estado"	Sim, um objeto JSON representa o estado de cada componente.	Sim, utilizando Hooks como o hook **useState**.

O Link 1.1 mostra uma introdução ao sistema de Hooks do React.

Link 1.1
https://reactjs.org/docs/hooks-intro.html

Neste material, estudaremos os principais aspectos sobre a definição de componentes React utilizando classes.

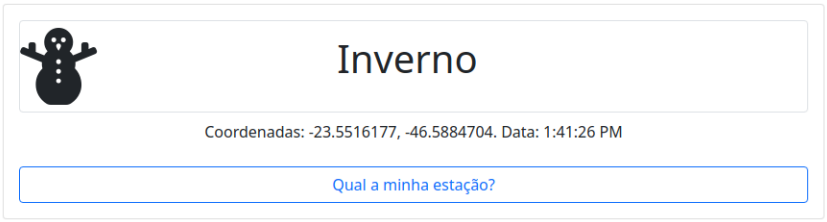
2 Desenvolvimento

A aplicação que desenvolveremos determina a estação climática atual em função da localização do usuário e da data em que ela é acessada. A Tabela 2.1 mostra os critérios que utilizaremos para determinar a estação climática.

Hemisfério/Data	21 de junho a 23 de setembro	24 de setembro a 21 de dezembro	22 de dezembro a 20 de março	21 de março a 20 de junho
Norte	Verão	Outono	Inverno	Primavera
Sul	Inverno	Primavera	Verão	Outono

A sua interface gráfica aparece na Figura 2.1.

Figura 2.1



2.1 (Criando a aplicação e ajustando as dependências) Utilizando um terminal vinculado ao seu workspace, use

npx create-react-app estacao-climatica

para criar a aplicação. Use

cd estacao-climatica

para navegar até o diretório recém-criado e

code .

para abrir uma instância do VS Code vinculada a ele. Após clicar Terminal >> New Terminal no VS code e abrir um terminal vinculado a ele, use

npm start

para colocar a aplicação em funcionamento. Uma aba de seu navegador padrão deverá ser aberta, fazendo uma requisição a **localhost:3000**.

Uma das dependências da aplicação é o Bootstrap. Use

npm install bootstrap

para fazer a sua instalação. Também utilizaremos os ícones FontAwesome. Para isso, adicione um CDN ao arquivo **public/index.html**, como destaca o Bloco de Código 2.1.1.

Bloco de Código 2.1.1

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta
      name="description"
      content="Web site created using create-react-app"
    />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <!--
      manifest.json provides metadata used when your web app is installed on a
      user's mobile device or desktop. See https://developers.google.com/web/fundamentals/web-app-manifest/
    -->
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/5.15.4/css/all.min.css" integrity="sha512-1ycn6IcaQQ40/MKBW2W4Rhis/DbILU74C1vSrLJxCq57o941Ym01SwNsOMqvEBFlcgUa6xLiPY/NS5R+E6ztJQ==" crossorigin="anonymous" referrerpolicy="no-referrer" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>

  </body>
</html>
```

Apague todos os arquivos existentes na pasta **src.** Ainda na pasta **src**, crie um arquivo chamado **index.js**. O Bloco de Código 2.1.1 mostra o seu conteúdo inicial.

Bloco de Código 2.1.1

```
import 'bootstrap/dist/css/bootstrap.min.css'
import React from 'react'
import ReactDOM from 'react-dom'

export default function App() {
   return (
      <div>
         Meu app
      </div>
   )
}

ReactDOM.render(
   <App />,
   document.querySelector("#root")
)
```

2.2 (Detectando a localização do usuário) Os principais navegadores da atualidade implementam uma API denominada "Geolocation". Ela permite, entre muitas outras coisas, obter a localização do usuário. Visite o Link 2.2.1 para ler a sua documentação disponível no MDN.

Link 2.2.1
https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API

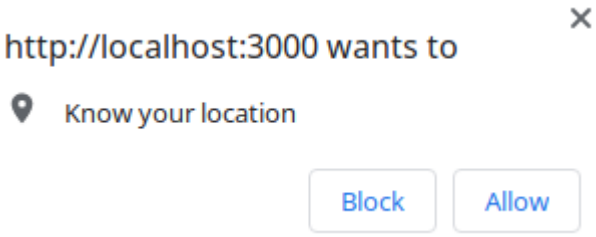
Faça um primeiro teste como no Bloco de Código 2.2.1, em que utilizamos a API para obter a localização do usuário e especificamos uma função que será executada quando a localização estiver disponível.

Bloco de Código 2.2.1

```
import 'bootstrap/dist/css/bootstrap.min.css'
import React from 'react'
import ReactDOM from 'react-dom'

export default function App() {

  window.navigator.geolocation.getCurrentPosition(
     (position) => console.log(position)
  )
   return (
      <div>
         Meu app
      </div>
   )
}

ReactDOM.render(
   <App />,
   document.querySelector("#root")
)
```

O seu navegador deverá exibir um simples *alert* que pergunta se você autoriza o acesso à sua localização, como na Figura 2.2.1.

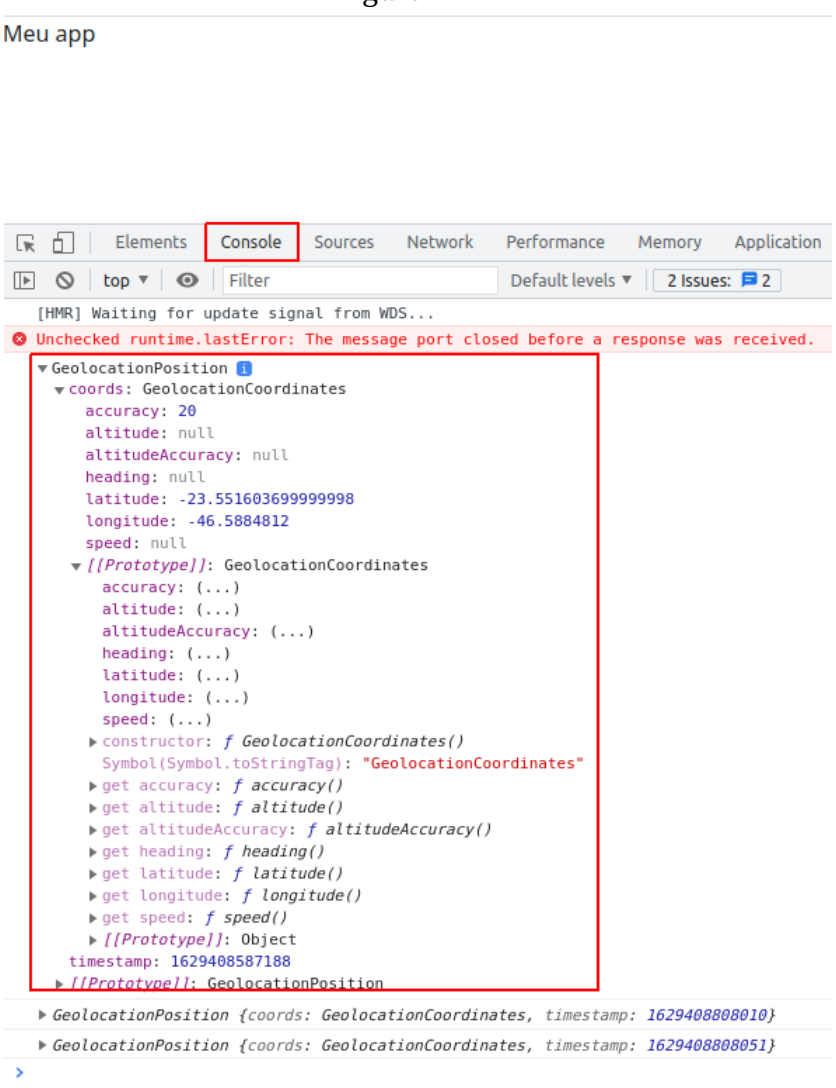
Figura 2.2.1



Clique **Allow** para que a aplicação possa exibir a sua localização real.

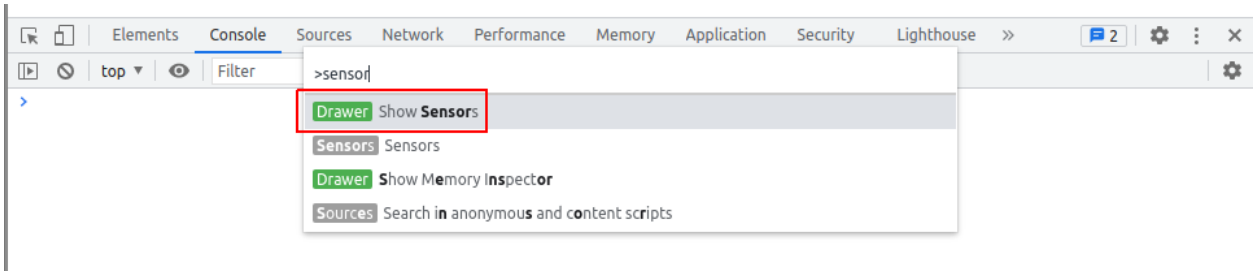
Abra o Chrome Dev Tools (CTRL + SHIFT + I) e clique na aba console para visualizar o resultado, que também é exibido pela Figura 2.2.2.

Figura 2.2.2



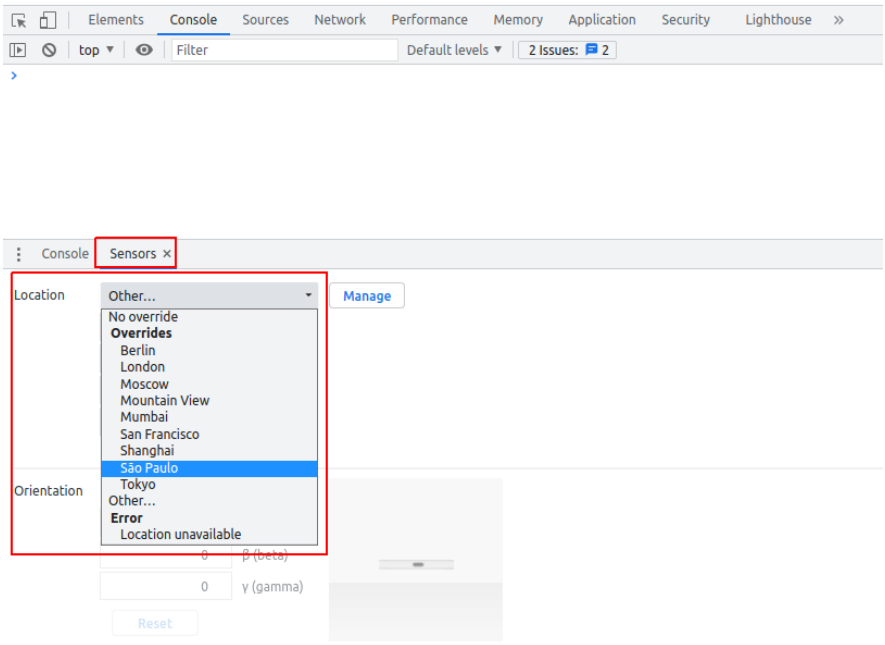
Também é possível utilizar localizações simuladas pelo próprio navegador. Para isso, mantenha o foco no console do Chrome Dev Tools e aperte CTRL + SHIFT + P. Na tela que aparece, digite **sensors**. Escolha a opção **Show sensors**, como ilustra a Figura 2.2.3.

Figura 2.2.3



Na opção associada a “Location”, escolha a opção desejada, como mostra a Figura 2.2.4.

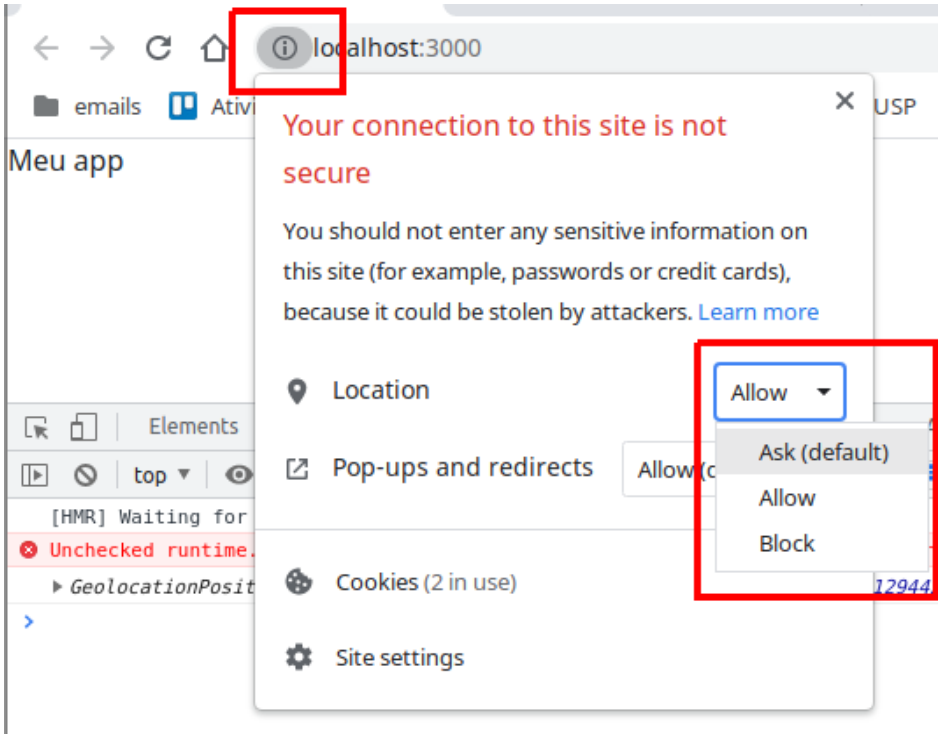
Figura 2.2.4



Caso deseje voltar atrás, basta voltar a este menu e escolher a opção **No override**.

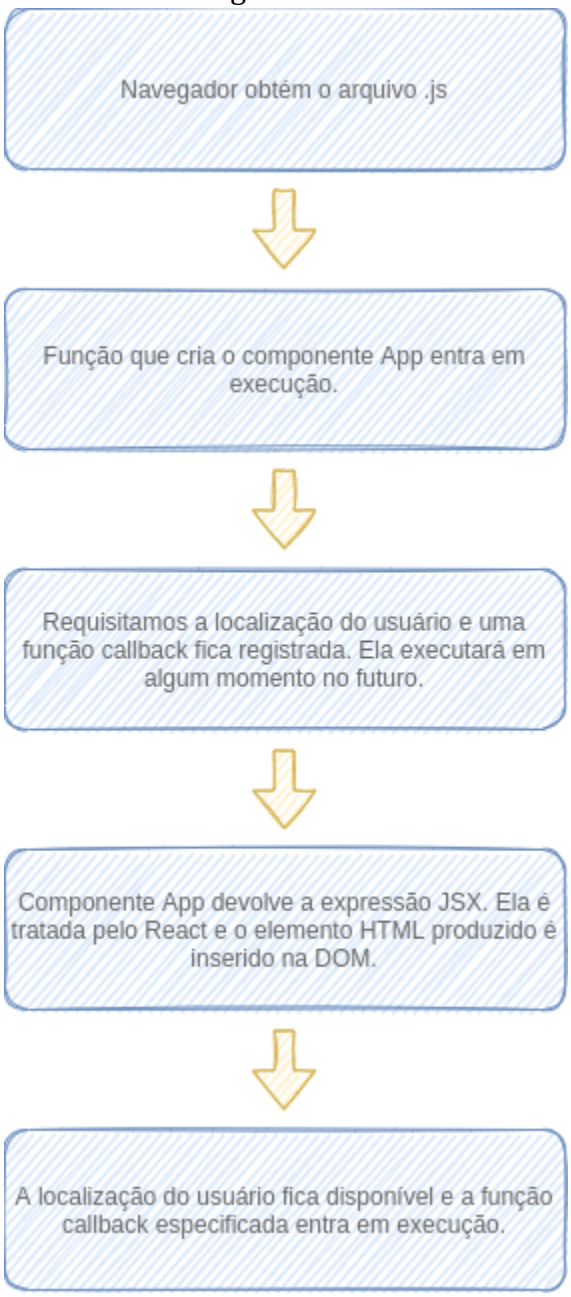
Uma vez que tenha clicado **Allow**, permitindo portando o acesso à sua localização, o navegador lembrará desta escolha e não perguntará novamente nas próximas vezes em que acessar esta página. Para reconfigurar o navegador de modo que ele solicite permissão para acesso à localização novamente, basta clicar no ícone que fica ao lado da barra de navegação do navegador, como mostra a Figura 2.2.5.

Figura 2.2.5



2.3 (Incluindo dados da localização, como latitude e longitude, na expressão JSX. E agora??) Neste ponto, desejamos incluir informações referentes à localização do usuário na expressão JSX devolvida pelo componente, afinal, nosso objetivo é visualizá-las. Entretanto, a operação é realizada de maneira **assíncrona** e os resultados são entregues em uma função **callback**, o que inviabiliza o acesso a eles na expressão JSX diretamente. A Figura 2.3.1 ilustra a sequência das operações que foram executadas.

Figura 2.3.1



Este é um cenário em que, antes de a versão **16.8** do React ser lançada, a qual traz o mecanismo conhecido como **Hooks**, componentes definidos por meio de classes precisavam ser utilizados. A seguir, veremos como fazê-lo. No futuro, trataremos do uso de Hooks.

2.4 (Reescrevendo o componente funcional utilizando uma classe) Quando um componente é definido por meio de uma classe, podemos realizar diversas tarefas. Entre elas, podemos dizer ao componente que ele precisa se atualizar – **renderizar novamente** - mediante determinados eventos, causando alterações naquilo que é apresentado ao usuário. O componente que criaremos terá, portanto, as seguintes características.

- Será definido por meio de uma classe
- A classe que o define deverá herdar de **React.Component**
- Definirá um método chamado **render**. Ele é responsável por produzir a expressão JSX de interesse.

Começamos removendo a definição do componente funcional App. A seguir, escrevemos a nova definição, usando uma classe. Veja o Bloco de Código 2.4.1.

Bloco de Código 2.4.1

```
import 'bootstrap/dist/css/bootstrap.min.css'
import React from 'react'
import ReactDOM from 'react-dom'

class App extends React.Component {
    render(){
        return (
            <div>
                Meu app
            </div>
        )
    }
}

ReactDOM.render(
    <App />,
    document.querySelector("#root")
)
```

Visite localhost:3000 agora, atualize a página e certifique-se de que a aplicação está funcionando.

2.5 (Definindo o estado do componente) Componentes baseados em classes possuem **estado**. Trata-se de um simples objeto JSON envolvido em diferentes situações de interesse. As suas principais características são as seguintes:

- Em versões anteriores à versao 16.8 do React, a noção de estado somente podia ser associada a componentes definidos por meio de classes.

- O estado de um componente é um simples objeto JSON que possui dados que são de interesse ao componente, como dados que serão exibidos na tela, por exemplo.

- Quando o estado de um componente é atualizado, o componente é renderizado novamente automaticamente e muito rapidamente.

- O estado de um componente deve ser criado quando ele é instanciado, o que pode ser feito em seu construtor.

- A função **setState deve** ser utilizada para atualizar o estado de um componente.

O Bloco de Código 2.5.1 mostra como definir o estado de nossa aplicação. Em seu construtor. Ele tem as seguintes propriedades:

- latitude
- longitude
- estacao
- data
- icone

Bloco de Código 2.5.1

```
…
class App extends React.Component {

   constructor(props) {
       super(props)
       this.state = {
           latitude: null,
           longitude: null,
           estacao: null,
           data: null,
           icone: null
       }

   }
...
```

2.6 (A função que determina a estação climática) A função exibida pelo Bloco de Código 2.6.1 mostra uma forma para se obter a estação climática atual – de maneira simples, desconsiderando possíveis variações de datas – em função da data atual e da latitude.

Nota. Estamos considerando que localizações com valor de latitude menor do que zero estão no hemisfério Sul. As demais estão no hemisfério Norte.

Bloco de Código 2.6.1

```
…
class App extends React.Component {

   constructor(props) {
      ...

   }
...
   obterEstacao = (data, latitude) => {
      const anoAtual = data.getFullYear()
      //new Date(ano, mês(0 a 11), dia(1 a 31))
      //21/06
      const d1 = new Date(anoAtual, 5, 23)
      //24/09
      const d2 = new Date(anoAtual, 8, 24)
      //22/12
      const d3 = new Date(anoAtual, 11, 22)
      //21/03
      const d4 = new Date(anoAtual, 2, 21)
      const sul = latitude < 0;
      if (data >= d1 && data < d2)
         return sul ? 'Inverno' : 'Verão'
      if (data >= d2 && data < d3)
         return sul ? 'Primavera' : 'Outono'
      if (data >= d3 || data < d1)
         return sul ? 'Verão' : 'Inverno'
      return sul ? 'Outono' : 'Primavera'
   }
...
```

2.7 (Um objeto JSON que faz o mapeamento entre ícones e estações climáticas) Dada uma estação climática, desejamos escolher um ícone relacionado a ela para exibir. Para tal, vamos definir um objeto JSON responsável pelo mapeamento. Veja o Bloco de Código 2.7.1.

Bloco de Código 2.7.1

```
…
obterEstacao = (data, latitude) => {
      ...
    }
icones = {
        'Primavera': 'fa-seedling',
        'Verão': 'fa-umbrella-beach',
        'Outono': 'fa-tree',
        'Inverno': 'fa-snowman'
}
...
```

2.8 (Obtendo localização, data e ícone a ser exibido) Quando o usuário interagir com a aplicação, desejamos

= Consultar a sua localização atual e registrar uma função callback que será executada quando a localização estiver disponível
- Extrair a data atual do sistema
- Definir a estação climática atual em função da latitude e data atual
- Escolher o ícone apropriado para a estação climática definida

A função do Bloco de Código 2.8.1 se encarrega disso.

Bloco de Código 2.8.1

```
...
    icones = {
        'Primavera': 'fa-seedling',
        'Verão': 'fa-umbrella-beach',
        'Outono': 'fa-tree',
        'Inverno': 'fa-snowman'
    }

    obterLocalizacao = () => {
        window.navigator.geolocation.getCurrentPosition(
            (posicao) => {
                let data = new Date()
                let estacao = this.obterEstacao(data, posicao.coords.latitude);
                let icone = this.icones[estacao]
                console.log(icone)
                this.setState(
                    {
                        latitude: posicao.coords.latitude,
                        longitude: posicao.coords.longitude,
                        estacao: estacao,
                        data: data.toLocaleTimeString(),
                        icone: icone
                    }
                )
            }
        )
    }
...
```

2.9 (A expressão JSX produzida pelo componente) Cabe à função **render** produzir a expressão JSX que define a aparência visual do componente. O Bloco de Código 2.9.1 mostra o primeiro passo, que trata da responsividade com classes Bootstrap. Repare nos comentários.

Bloco de Código 2.9.1

```
render() {
      return (
         // responsividade, margem acima
         <div className="container mt-2">
            {/* uma linha, conteúdo centralizado, display é flex */}
            <div className="row justify-content-center">
            {/* oito colunas das doze disponíveis serão usadas para telas médias em diante */}
               <div className="col-md-8">

               </div>
            </div>
         </div>
      )
   }
```

Os resultados serão exibidos em um cartão do Bootstrap comum, como define o Bloco de Código 2.9.2.

Bloco de Código 2.9.2

```
...
<div className="col-md-8">
        {/* um cartão Bootstrap */}
         <div className="card">
            {/* o corpo do cartão */}
             <div className="card-body">

         </div>
</div>
...
```

O primeiro elemento que o cartão exibe contém o ícone e o nome da estação. Veja o Bloco de Código 2.9.3.

Bloco de Código 2.9.3

```
<div className="card-body">
        {/* centraliza verticalmente, margem abaixo */}
         <div className="d-flex align-items-center border rounded mb-2"
             style={{ height: '6rem' }}>
                {/* ícone obtido do estado do componente */}
                 <i className={`fas fa-5x ${this.state.icone}`}></i>
                 {/* largura 75%, margem no à esquerda (start), fs aumenta a fonte */}
                  <p className=" w-75 ms-3 text-center fs-1">{this.state.estacao}</p>
         </div>
</div>
```

O elemento que o cartão exibe logo abaixo do primeiro elemento é um texto que é definido por meio de uma **expressão condicional**. Ela funciona assim: caso a localização já tenha sido obtida e, portanto, a latitude já tenha sido armazenada no estado do componente, o elemento mostra as coordenadas do local onde o usuário se encontra e a data atual do sistema. Caso contrário, o texto exibido contém instruções sobre como utilizar a aplicação. Veja a definição do elemento no Bloco de Código 2.9.4.

Bloco de Código 2.9.4

```
...
<div className="card-body">
      {/* centraliza verticalmente, margem abaixo */}
      <div className="d-flex align-items-center border rounded mb-2"
              style={{ height: '6rem' }}>
            {/* ícone obtido do estado do componente */}
            <i className={`fas fa-5x ${this.state.icone}`}></i>
            {/* largura 75%, margem no à esquerda (start), fs aumenta a fonte */}
             <p className=" w-75 ms-3 text-center fs-1">{this.state.estacao}</p>
       </div>
        <div>
          <p className="text-center">
            {/* renderização condicional */}
            {
               this.state.latitude ?
               `Coordenadas: ${this.state.latitude}, ${this.state.longitude}. Data: ${this.state.data}`
               :
               'Clique no botão para saber a sua estação climática'
            }
           </p>
         </div>
</div>
...
```

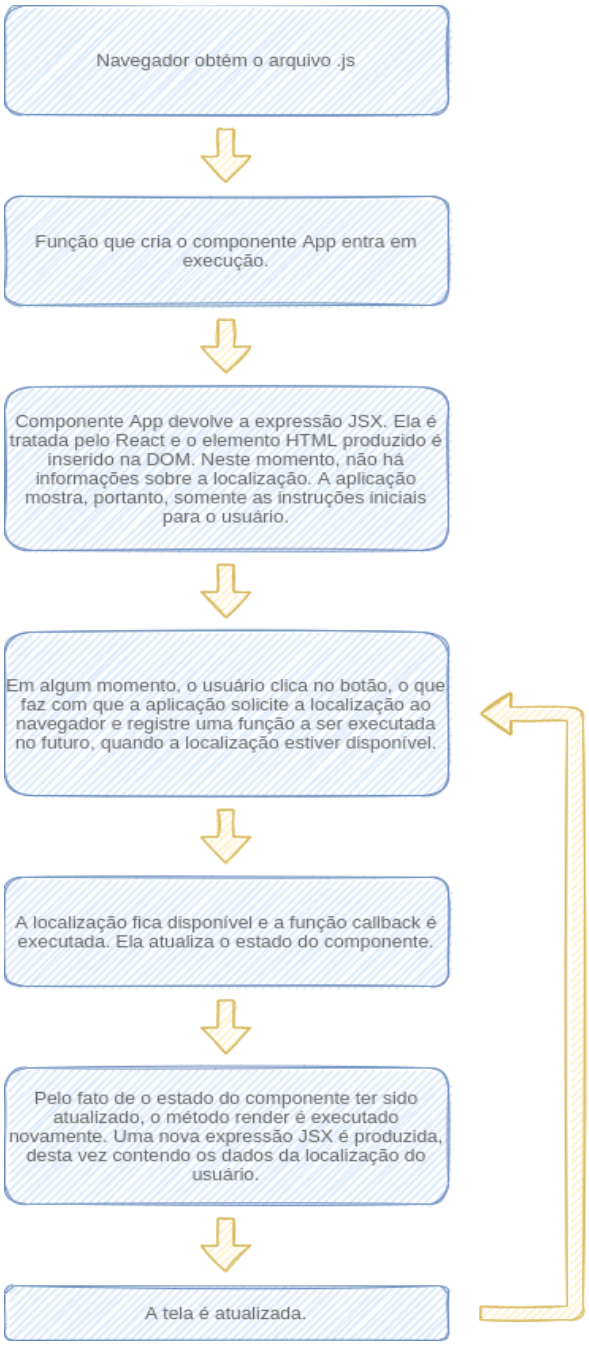
Finalmente, adicionamos um botão para que o usuário possa colocar a aplicação em funcionamento, como no Bloco de Código 2.9.5.

Bloco de Código 2.9.5

```
...
<div className="card-body">
      {/* centraliza verticalmente, margem abaixo */}
      <div className="d-flex align-items-center border rounded mb-2"
              style={{ height: '6rem' }}>
            {/* ícone obtido do estado do componente */}
            <i className={`fas fa-5x ${this.state.icone}`}></i>
            {/* largura 75%, margem no à esquerda (start), fs aumenta a fonte */}
             <p className=" w-75 ms-3 text-center fs-1">{this.state.estacao}</p>
       </div>
        <div>
          <p className="text-center">
            {/* renderização condicional */}
            {
               this.state.latitude ?
               `Coordenadas: ${this.state.latitude}, ${this.state.longitude}. Data: ${this.state.data}`
               :
               'Clique no botão para saber a sua estação climática'
            }
           </p>
         </div>
          {/* botão azul (outline, 100% de largura e margem acima) */}
          <button onClick={this.obterLocalizacao}
                 className="btn btn-outline-primary w-100 mt-2">
                  Qual a minha estação?
          </button>
</div>
...
```

A Figura 2.9.1 ilustra a sequência de fatos importantes que ocorrem conforme a aplicação é utilizada. Repare como **a função render é chamada a cada vez que o estado do componente é atualizado**.

Figura 2.9.1


Navegador obtém o arquivo .js
Função que cria o componente App entra em execução.
Componente App devolve a expressão JSX. Ela é tratada pelo React e o elemento HTML produzido é inserido na DOM. Neste momento, não há informações sobre a localização. A aplicação mostra, portanto, somente as instruções iniciais para o usuário.
Em algum momento, o usuário clica no botão, o que faz com que a aplicação solicite a localização ao navegador e registre uma função a ser executada no futuro, quando a localização estiver disponível.
A localização fica disponível e a função callback é executada. Ela atualiza o estado do componente.
Pelo fato de o estado do componente ter sido atualizado, o método render é executado novamente. Uma nova expressão JSX é produzida, desta vez contendo os dados da localização do usuário.
A tela é atualizada.

2.10 (Tratando erros) O que ocorre com a aplicação caso o navegador não possa lhe entregar a localização do usuário? Talvez ele não tenha suporte a essa API. Talvez não exista um mecanismo de detecção de localização disponível no dispositivo do usuário. Talvez o usuário tenha negado o acesso `à sua localização. Idealmente, no mínimo, a aplicação exibe mostra ao usuário um feedback textual explicando que um erro aconteceu.

- Para fazer esse tratamento, começamos adicionando um campo no estado do componente que armazenará as possíveis mensagens de errro. Veja o Bloco de Código 2.10.1.

Bloco de Código 2.10.1

```
...
constructor(props) {
        super(props)
        this.state = {
            latitude: null,
            longitude: null,
            estacao: null,
            data: null,
            icone: null,
            mensagemDeErro: null
        }

}
...
```

- A seguir, especificamos uma segunda função callback, também enviada como argumento para a função que obtém a localização do usuário. Em caso de sucesso, a primeira - que já havíamos definido - é executada. Caso contrário, a segunda entra em execução. Ela exibe a mensagem de erro como um log para o desenvolvedor (não é boa prática exibir os detalhes internos do aplicativo para o usuário) e uma mensagem para o usuário que explica que um erro aconteceu e que ele pode tentar mais tarde. Veja o Bloco de Código 2.10.2.

Bloco de Código 2.10.2

```
…
obterLocalizacao = () => {
      window.navigator.geolocation.getCurrentPosition(
         (posicao) => {
            let data = new Date()
            let estacao = this.obterEstacao(data, posicao.coords.latitude);
            let icone = this.icones[estacao]
            console.log(icone)
            this.setState(
               {
                  latitude: posicao.coords.latitude,
                  longitude: posicao.coords.longitude,
                  estacao: estacao,
                  data: data.toLocaleTimeString(),
                  icone: icone
               }
            )
         },
         (erro) => {
            console.log(erro)
            this.setState({mensagemDeErro: `Tente novamente mais tarde`})
         }
      )
 }
...
```

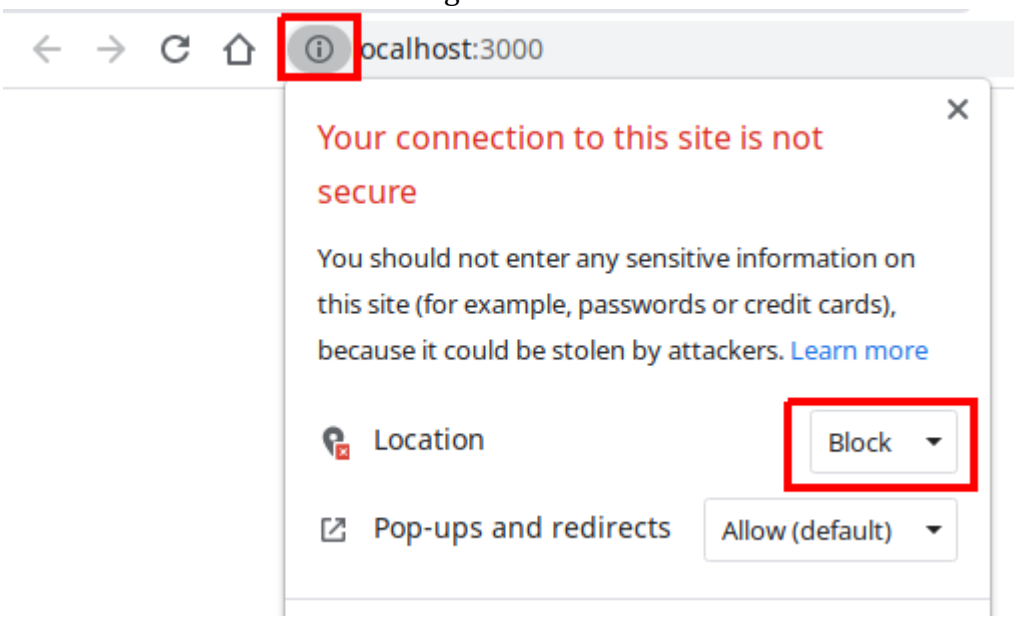
- Como a função de erro atualiza o estado do componente, ela faz com que a função render seja chamada uma vez mais. Ela passa a utilizar a mensagem de erro, caso exista. Veja o Bloco de Código 2.10.3. Repare que utilizamos uma expressão condicional aninhada.

Bloco de Código 2.10.3

```
...
 <p className="text-center">
                                {/* renderização condicional */}
                                {
                                   this.state.latitude ?
                                      `Coordenadas: ${this.state.latitude}, ${this.state.longitude}.
                                            Data: ${this.state.data}`
                                   :
                                   this.state.mensagemDeErro ?
                                      `${this.state.mensagemDeErro}`
                                   :
                                      'Clique no botão para saber a sua estação climática'

                                }
 </p>
...
```

- Para testar, remova a permissão de acesso à localização no navegador e atualize a página. Veja a Figura 2.10.1.

Figura 2.10.1



- Pode ser de interesse averiguar a estrutura do estado da aplicação conforme o usuário interage com ela. Para isso, exiba o estado no começo do método render, como no Bloco de Código 2.10.4.

Bloco de Código 2.10.4

```
...
 render() {
       console.log(this.state)
       return (
...
```

A seguir, execute a aplicação e visualize o log no Chrome Dev Tools (CTRL + SHIFT + I), aba console. O resultado esperado é parecido com o que exibem as figuras 2.10.2 e 2.10.3. Repare que, conforme o estado é atualizado, propriedades previamente existentes nunca são removidas.

Figura 2.10.2 - Aqui o usuário deu permissão de acesso à localização

Figura 2.10.3 - Aqui o usuário negou acesso à sua localização

Referências

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