INSTITUTO DE GESTÃO E TECNOLOGIA
DA INFORMAÇÃO

React II

Danilo Ferreira e Silva

2022

React II

Danilo Ferreira e Silva

Sumário

Capítulo 1. Introdução 5

1.1. Visão geral do módulo 5

1.2. Ferramentas de desenvolvimento 6

1.3. Back End da aplicação 6

Capítulo 2. TypeScript 8

2.1. Verificação de tipos 8

2.2. Tipos primitivos, arrays e objetos 9

2.3. Interfaces 9

2.4. A palavra-chave type 10

2.5. Funções 11

2.6. Tipos como parâmetros 11

2.7. Classes 11

Capítulo 3. Interface com Material-UI 12

3.1. Instalação 12

3.2. Uso do Material-UI 12

Capítulo 4. Adicionando estado e comportamento 14

Capítulo 5. Roteamento com react-router 15

5.1. Rotas com parâmetros 16

5.2. Acesso ao objeto history 17

Capítulo 6. Modularizando a aplicação 18

Capítulo 7. Acesso direto ao DOM com useRef 19

Capítulo 8. Autenticação 21

Capítulo 9. Context 22

9.1. Criando um Context 22

9.2. Utilizando o Context 22

Capítulo 10. Otimização de aplicações 24

10.1. useMemo 24

10.2. React.memo 25

10.3. useCallback 26

Capítulo 11. useReducer 27

Capítulo 12. Hooks customizados 28

Capítulo 13. Class Components 29

13.1. Ciclo de vida do componente 30

Capítulo 14. Redux 31

14.1. Instalação do Redux 31

14.2. Definição do reducer e store 31

14.3. Uso nos componentes React 32

Referências 34

Capítulo 1. Introdução

Este capítulo dá uma visão geral do Módulo React II e descreve as ferramentas de desenvolvimento necessárias para acompanhar o curso. É importante destacar que esta apostila não contém todo o conteúdo das videoaulas, mas apenas informações de referência rápida, dando apoio as mesmas.

1.1. Visão geral do módulo

Neste módulo, construiremos uma aplicação React de média complexidade, introduzindo uma série de novos conhecimentos ao longo do processo, entre elas:

- Linguagem TypeScript e seu uso em aplicações React;
- Uso da biblioteca de componentes Material UI;
- Implementação de roteamento *client-side* nas transições de telas do *Front End*, usando a biblioteca react-router.
- Noções de modularização da aplicação;
- Acesso direto ao DOM com *hook* useRef;
- Comunicação com Back End com autenticação;
- A API Context;
- Otimização de aplicações com React.memo e *hooks* useMemo e useCallback;
- *hook* useReducer;

- Criação de *hooks* customizados;
- E a biblioteca Redux.

Como aplicação de exemplo, construiremos uma ferramenta de agenda, que permite visualizar o calendário, adicionar eventos e organizá-los em agendas. Nossa aplicação será chamada de Agenda React.

1.2. Ferramentas de desenvolvimento

Para acompanhar o módulo, é importante ter instalado as seguintes ferramentas já utilizadas em módulos anteriores:

- Visual Studio Code.
- Node.js.
- Navegador Google Chrome.

Ao longo das videoaulas, também utilizamos as extensões do Google Chrome React Developer Tools e Redux DevTools. Além disso, foi utilizada a extensão do VSCode Prettier - Code formatter.

1.3. Back End da aplicação

Para focarmos na construção do Front End em React, utilizaremos um Back End pronto e construído em Node.js, fornecido pelo professor junto com o código fonte utilizado nas videoaulas.

Antes de executá-lo, entre no diretório Back End e execute o comando:

```
npm install
```

Em seguida, é possível iniciar o Back End de duas maneiras, sem autenticação (até o capítulo 7):

```
npm start -- noauth
```

Ou com autenticação (capítulo 8 em diante):

```
npm start
```

Capítulo 2. TypeScript

TypeScript é uma extensão da linguagem JavaScript que adiciona um sistema de verificação de tipos em tempo de compilação. Ou seja, em JavaScript, erros de tipo ocorrem durante a execução caso não sejam devidamente tratados, em TypeScript, erros de tipo são encontrados pelo compilador TypeScript. Com isso, o desenvolvimento se torna menos propenso a erros, trazendo maior facilidade de manutenção.

O React possui suporte oficial ao uso de TypeScript para a codificação da aplicação. Usaremos tal linguagem ao longo deste módulo, portanto, precisamos de noções básicas desta linguagem. Esta seção não pretende ser um guia extensivo de TypeScript, mas apenas uma breve introdução. Mais detalhes serão apresentados ao longo do curso, conforme necessário.

2.1. Verificação de tipos

Todo código JavaScript é também um código TypeScript. Ou seja, TypeScript apenas adiciona novos recursos à linguagem. O mais importante desses recursos é a capacidade de declarar tipos em variáveis, parâmetros de funções, retornos de funções e em várias outras situações. Por exemplo, considere a função abaixo:

```
function soma(a: number, b: number): number {
    return a + b;
}
```

Observe que os parâmetros da função são anotados com o tipo number, assim como o tipo de retorno da função é number. Dessa forma, se tentarmos chamar tal função passando parâmetros do tipo incorreto, por exemplo:

```
soma(3, 'olá'); // erro!
```

O compilador acusaria um erro, pois é esperado que o segundo parâmetro seja number, e não string. De forma semelhante, não podemos atribuir um valor do tipo incorreto para uma variável quando declaramos o seu tipo. O Visual Studio Code está preparado para lidar com código TypeScript, e exibe erros de tipo dentro do próprio editor.

2.2. Tipos primitivos, arrays e objetos

Podemos declarar tipos primitivos, arrays e objetos em TypeScript, conforme o exemplo a seguir.

```
// Podemos declarar tipos primitivos...
const varNumber: number = 1;
const varBoolean: boolean = false;
const varString: string = "react";
const varUndefined: undefined = undefined;

// Arrays...
const varArrayNumber: number[] = [1, 2, 3];
const varArrayString: string[] = ["a", "b", "c"];

// e Objetos.
const varObjeto: { x: number, y: number } = { x: 1, y: 2 };
```

2.3. Interfaces

Podemos declarar uma interface para descrever a estrutura de um objeto, por exemplo:

```
// Com interfaces podemos dar um nome para um tipo
interface Aluno {
  matricula: number,
  nome: string,
  dataNascimento?: string // campo opcional
}

let varAluno: Aluno = {
```

```
  matricula: 123,
  nome: "João"
}
```

Cabe ressaltar que interfaces existem apenas em TypeScript, elas desaparecem quando o código é transformado em JavaScript, seu propósito, portanto, é apenas a verificação de tipos.

2.4. A palavra-chave type

Além de declarar interfaces, podemos usar a palavra-chave *type* para criar um tipo nomeado, a partir de uma expressão que define um tipo. É possível construir tipos usando os operadores de união ou interseção, conforme o exemplo a seguir:

```
// Também podemos criar type alias
type MeuTipo = string;


// Podemos definir tipos como união de dois ou mais tipos
let varNumOuStr: number | string = 3;
varNumOuStr = "x";

// Outro exemplo, notem que literais podem ser usados como tipos
type Alinhamento = "esquerda" | "direita" | "centro";
let varAlinhamento: Alinhamento = "direita";


// Também podemos usar a interseção
interface Programador {
  linguagemFavorita: string;
}

let varAlunoProgramador: Aluno & Programador = {
  matricula: 123,
  nome: "João",
  linguagemFavorita: "TypeScript"
};
```

2.5. Funções

Também podemos definir tipos que descrevem funções, por exemplo:

```
// Podemos definir tipos para funções
type OperadorNumerico = (n1: number, n2: number) => number;

const opDivisao: OperadorNumerico = (n1, n2) => n1 / n2;
```

2.6. Tipos como parâmetros

É possível receber tipos como parâmetros para definir, por exemplo, o tipo de retorno de uma função de acordo com o tipo do parâmetro:

```
// Funções podem receber tipos como parâmetros
function pegaMaior<T>(a: T, b: T): T {
  if (a > b) {
    return a;
  } else {
    return b;
  }
}
```

2.7. Classes

TypeScript, assim como JavaScript moderno, aceita a sintaxe de declaração de classes, facilitando a programação orientada a objetos. Adicionalmente, podemos definir tipos para os membros da classe:

```
class Retangulo {
    altura: number;
    largura: number;
    calculaArea(): number {
        // ...
    }
}
```

Capítulo 3. Interface com Material-UI

Neste capítulo, começamos a construir a interface da aplicação Agenda React usando Material-UI. Material-UI é uma biblioteca de componentes para construção de interfaces baseada no padrão de Material Design da Google.

3.1. Instalação

A biblioteca pode ser instalada via npm, usando o seguinte comando:

```
npm install @material-ui/core --save
```

Além de instalar o pacote via npm, é interessante incluir as folhas de estilo abaixo no arquivo index.html da aplicação. O primeiro deles inclui a fonte Roboto, que é recomendada para uso com a Material-UI. O segundo inclui o pacote de ícones Material Icons.

```
<link
  rel="stylesheet"
  href="https://fonts.googleapis.com/css?family=Roboto:300,400,500,700&display=swap"
/>
<link rel="stylesheet" href="https://fonts.googleapis.com/icon?family=Material+Icons"
 />
```

3.2. Uso do Material-UI

Cada componente disponibilizado pela biblioteca é autocontido e pode ser importado e utilizado individualmente. Os componentes já possuem estilização sem a necessidade de importar uma folha de estilos, pois, abaixo, temos um exemplo de uso do componente Button:

```
import Button from "@material-ui/core/Button";
```

```
function App() {
  return <Button color="primary">Hello World</Button>;
}
```

Para customizar a estilização dos componentes, podemos usar a mesma solução de CSS em JavaScript do Material-UI. Para isso, devemos usar a função makeStyles, passando os estilos desejados como parâmetro, e usar o valor de retorno da mesma com um hook, dentro do componente:

```
import Button from "@material-ui/core/Button";
import { makeStyles } from "@material-ui/core/styles";

const useStyles = makeStyles({
  customClass: {
    color: "red",
  },
});

function App() {
  const classes = useStyles();
  return <Button className={classes.customClass}>Hello World</Button>;
}
```

Capítulo 4. Adicionando estado e comportamento

Neste capítulo, adicionamos comportamento à aplicação Agenda React, gerando o calendário e buscando dados do Back End. Para isso, foram utilizados hooks, como useEffect e a API fetch para busca os dados. Portanto, não introduzimos nenhum assunto novo neste capítulo.

Capítulo 5. Roteamento com react-router

Quando desenvolvemos aplicações React, que tipicamente consistem em uma única página (Single Page Application), é importante que utilizemos roteamento do lado do cliente para registrar as mudanças de tela, como entradas no histórico do navegador. Dessa forma, teremos suporte ao voltar/avançar no navegador. Para implementar o roteamento, utilizaremos a biblioteca react-router.

Para usar o react-router, devemos utilizar os seguintes componentes fornecidos pela biblioteca:

- **BrowserRouter**: Devemos englobar os demais componentes dentro deste, para que eles tenham acesso às funcionalidades da API.
- **Route**: Componente que exibe seu conteúdo de acordo apenas quando a rota especificada, como prop, está ativa. Este componente é a base do react-router.
- **Switch**: Exibe o primeiro componente **Route** ativo dentro do dele.
- **Redirect**: Redireciona para uma rota especificada.
- **Link**: Gera um link para uma rota.

Abaixo, temos um exemplo de uso em uma aplicação simplificada.

```
import { BrowserRouter, Switch, Route, Redirect } from "react-router-
dom";

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route path="/rota1">
          <Componente1 />
        </Route>
```

```
        <Route path="/rota2">
          <Componente2 />
        </Route>
        <Redirect to={{ pathname: "/rota1" }} />
      </Switch>
    </BrowserRouter>
  );
}
```

Neste exemplo, definimos duas rotas (/rota1 e /rota2), além de um redirecionamento para /rota1 caso a URL não case com nenhuma das duas rotas.

Podemos gerar um link para uma rota usando o componente **Link**.

```
import { Link } from "react-router-dom";

function MeuComponente() {
  return (
    <div>
      <Link to="/rota1" />
    </div>
  );
}
```

5.1. Rotas com parâmetros

Uma rota pode ser composta por segmentos dinâmicos, que chamamos de parâmetros da rota. Por exemplo, podemos definir a rota `/calendar/:month`, onde *month* é um parâmetro.

Para acessar um parâmetro de rota em um determinado componente, devemos utilizar o *hook* useParams, conforme o exemplo a seguir:

```
import { useParams } from "react-router";

function MeuComponente() {
  const { month } = useParams<{ month: string }>();
}
```

Observem que, em TypeScript, devemos passar um tipo parametrizado na chamada **useParams**, descrevendo os campos esperados, pois o compilador não possui informações suficientes para inferi-los.

5.2. Acesso ao objeto history

Em certos casos, pode ser necessário utilizar o *hook* useHistory para ter acesso ao objeto history, que permite obter detalhes da rota atual, bem como manipular o histórico diretamente. Por exemplo, no código a seguir, chamamos a função history.push para navegar para outra rota:

```
import { useHistory } from "react-router-dom";

function MeuComponente() {
  const history = useHistory();
  return <button onClick={() => history.push("/outra-
rota")}>Navegar</button>;
}
```

Capítulo 6. Modularizando a aplicação

Neste capítulo, subdividimos o componente CalendarScreen em componentes menores. Em geral, devemos seguir as seguintes diretrizes ao modularizar uma aplicação.

- Procure criar componentes pequenos e com um objetivo claro. Subdivida o componente quando ele se tornar complexo.
- Quando o componente precisa de estado, e tal estado é usado apenas no próprio componente, armazene nele mesmo.
- Quando o estado precisa ser usado em mais de um componente, armazene-o em um que seja um ancestral em comum na hierarquia de componentes. Passe as informações do componente pai para o filho via props.
- Se um componente filho precisa enviar informações para o pai, ou seja, alterar o estado que está armazenado em um componente acima na hierarquia, passe uma função de *call-back* do pai para o filho.

Essas diretrizes básicas servem para a maioria das situações de modularização. No entanto, o maior desafio reside em saber quais funcionalidades colocar em cada componente, de forma que eles fiquem coesos, ou seja, internamente tratem de um mesmo assunto e, também, possuam baixo acoplamento com os demais componentes.

Capítulo 7. Acesso direto ao DOM com useRef

O *hook* useRef nos permite armazenar um valor que persiste durante toda a vida de um componente, assim como o useState. A grande diferença entre ambos é que mudanças no valor de useRef não fazem que o componente seja renderizado novamente.

Embora o useRef possa ser utilizado para variados propósitos, ele é mais frequentemente utilizado para armazenar referências a elementos, para que tenhamos acesso direto ao DOM. No React, todo elemento HTML aceita uma propriedade **ref** que pode receber um objeto criado com useRef. Ao renderizar o componente, a referência ao elemento correspondente do DOM é armazenada no objeto. O exemplo a seguir demonstra esse comportamento:

```
import { useRef } from "react";

function MeuComponente() {
  const meuRef = useRef<HTMLInputElement | null>(null);
  return (
    <div>
      <input ref={meuRef} />
      <button onClick={() => meuRef.current?.focus()}>Dá foco no campo</butto
n>
    </div>
  );
}
```

Nesse exemplo, usamos o objeto **meuRef** para referenciar o elemento input e chamar a função *focus* do mesmo quando pressionamos o botão. É importante ressaltar os seguintes pontos com relação ao código:

- useRef recebe como parâmetro o valor inicial. Como não temos acesso ao elemento input diretamente, precisamos inicializar com o valor null.

- Acessamos o valor armazenado por meio da propriedade current.

- useRef recebe o tipo parametrizado HTMLInputElement ou null. Isso é necessário, neste caso, pois se omitíssemos o tipo, o compilador inferiria o tipo como null, visto que esse é o valor inicial.

Em React é sempre bom privilegiar o código declarativo em detrimento de código imperativo, que chama diretamente funções de elemento no DOM. No entanto, em determinadas situações, isso é inevitável. Além do exemplo dado da função focus, podemos citar situações nas quais deseja-se fazer scroll no elemento (scrollIntoView) ou obter as dimensões de um elemento após a renderização, para cálculos mais avançados de layout.

Por fim, vale ressaltar que useRef também pode ser usado para armazenar valores arbitrários. Outro tipo de uso comum é utilizá-lo para armazenar um id gerado para o elemento, de forma que ele persista durante toda a vida do componente.

Capítulo 8. Autenticação

Neste capítulo, introduzimos autenticação à aplicação Agenda React. O Back End da aplicação, quando iniciado sem o parâmetro **noauth**, retorna o *status code* 401 para qualquer requisição feita sem autenticação. Para autenticar, devemos utilizar o *endpoint* **/auth/login**, passando e-mail e senha, criando, então, uma sessão. Utilizamos um *cookie* para representar e identificar a sessão do lado do cliente, como é comum em aplicações Web. A seguir, temos um resumo dos endpoints relacionados com a autenticação:

GET /auth/user	Obtém informações do usuário logado se houver sessão, caso contrário retorna erro 401.
POST /auth/login	Verifica e-mail/senha e cria uma sessão no Back End. O id da sessão é armazenado em um cookie HttpOnly.
POST /auth/logout	Destrói a sessão no Back End.

Para fins de teste, existe um único usuário na base, com e-mail *danilo@email.com* e senha *1234*.

É muito importante ressaltar que, quando desejamos enviar ou armazenar cookies ao fazer requisições HTTP com a API fetch, devemos passar a configuração `credentials: "include"`, conforme o exemplo a seguir:

```
function getCalendarsEndpoint(): Promise<ICalendar[]> {
  return fetch("http://localhost:8080/calendars", {
    credentials: "include",
  }).then(handleResponse);
}
```

Capítulo 9. Context

A API Context nos permite passar valores entre componentes sem precisar repassá-los via props. Isso é particularmente útil quando precisamos passar valores do topo da hierarquia de componentes para componentes que estão vários níveis abaixo. Na aplicação Agenda React, esse foi o caso das informações de usuário logado, que eram repassadas por vários componentes até chegarem naquele que realmente as utilizava.

9.1. Criando um Context

Antes de utilizar, precisamos criar um contexto com a função createContext. Essa função recebe um valor padrão a ser fornecido na falta de um Provider (ver a seguir). No código a seguir, demonstramos a criação de um contexto que fornece um objeto com a propriedade user e a função onSignOut:

```
import React from "react";

export const authContext = React.createContext<IAuthContext>({
  user: {
    name: "Anônimo",
    email: "",
  },
  onSignOut: () => {},
});
```

9.2. Utilizando o Context

Para utilizar o contexto, primeiro devemos utilizar o seu componente Provider associado, passando o valor. Os componentes que acessarão o valor do contexto devem ser descendentes do seu **Provider** na hierarquia de componentes. Para utilizar o valor, usamos o hook **useContext**, passando o contexto criado como parâmetro.

```
<authContext.Provider value={{ user, onSignOut }}>
```

```
  {/* componentes da aplicação, aqui dentro o contexto está disponível*/}
</authContext.Provider>
```

Assim como ocorre com useState, mudanças de valor no contexto disparam uma nova renderização dos componentes que o acessa por meio do **useContext**. Abaixo, temos um exemplo de uso o useContext:

```
import { useContext } from "react";
import { authContext } from "./authContext";

function MeuComponente() {
  const auth = useContext(authContext);
  return <div>{auth.user.name}</div>;
}
```

Capítulo 10. Otimização de aplicações

Uma das principais vantagens de utilizar React é que ele cuida da atualização dos elementos do DOM, enquanto precisamos cuidar apenas que nosso componente renderize o virtual DOM correto. O React renderiza os componentes sempre que houver mudanças de estado e, por padrão, todos os componentes descendentes são renderizados novamente. Normalmente, isso não é um problema, pois o algoritmo de comparação do virtual DOM é bastante rápido. No entanto, em determinadas situações, pode ser necessário evitar renderizações desnecessárias ou a execução de cálculos pesados.

10.1. useMemo

O hook useMemo pode ser usado para computar um valor apenas quando um conjunto de dependências informado sofrer alguma alteração. Considere o exemplo a seguir:

```
function MeuComponente() {
  const [valor1] = useState(1);
  const [valor2] = useState(2);
  const [valor3] = useState(3);

  const resultado = useMemo(() => {
    return fazCalculoPesadoQueUsaValor1eValor2(valor1, valor2);
  }, [valor1, valor2]);

  return <div>{/* conteúdo do componente ... */}</div>;
}
```

Com useMemo, a função passada como primeiro parâmetro só é executada se algum dos valores no array, passado como segundo parâmetro, alterar de uma renderização para a outra. Ou seja, neste caso específico, apenas quando valor1 ou valor2 mudar. Se não usarmos useMemo, toda renderização executaria o cálculo.

Quando o segundo parâmetro de useMemo é um array vazio, o código será executado apenas na primeira vez que o componente for renderizado, similarmente ao *hook* useEffect.

10.2. React.memo

A função React.memo recebe como parâmetro um componente (uma função) e retorna um componente otimizado que renderiza apenas quando seu props for alterado. Com isso, é possível evitar que toda uma árvore de componentes filhos deixe de ser renderizada ao alterar o estado do componente pai, obtendo ganhos de desempenho.

```
function Comp1(props: { n: number }) {
  return <div>{/* conteúdo do componente*/}</div>;
}

const Comp2 = React.memo(function (props: { n: number }) {
  return <div>{/* conteúdo do componente*/}</div>;
});

function MeuComponente() {
  const [valor1] = useState(1);
  const [valor2] = useState(2);

  return (
    <div>
      <Comp1 n={valor1} />
      <Comp2 n={valor2} />
    </div>
  );
}
```

No exemplo acima, Comp2 é renderizado apenas quando valor2 mudar, enquanto Comp1 é renderizado sempre que MeuComponente for renderizado (quando valor1 ou valor2 mudar).

É importante ressaltar que a comparação do props é feita de forma rasa, ou seja, o props é considerado igual se cada propriedade dele for igual ao comparador de igualdade (===). Isso é relevante, pois objetos, arrays e funções são comparadas por referência.

10.3. useCallback

O *hook* useCallback é semelhante ao useMemo, mas, em vez de executar uma função que computa o valor, ele retorna a própria função passada como parâmetro apenas quando o array de dependências alterar, caso contrário, retorna a mesma função retornada na renderização anterior.

Isso, muitas vezes, é importante para permitir que componentes com useMemo funcionem corretamente quando uma função é passada como prop. Como funções são comparadas por referência, a cada renderização, funções aninhadas no componente serão diferentes.

Capítulo 11. useReducer

O hook useReducer é uma alternativa ao useState para armazenar estado. A diferença entre eles é que useReducer recebe uma função responsável pelas transições de estado, conhecida como função *reducer*. A função *reducer* recebe como parâmetro o estado atual e um objeto action (que representa uma ação) e deve retornar o próximo estado. No exemplo a seguir, temos uma aplicação mínima que armazena um contador e permite duas ações: incrementar ou reiniciar o contador.

```
function reducer(state: number, action: { type: "increment" | "reset" })
{
  switch (action.type) {
    case "increment":
      return state + 1;
    case "reset":
      return 0;
    default:
      return state;
  }
}

function MeuComponente() {
  const [counter, dispatch] = useReducer(reducer, 0);
  return (
    <div>
      <button onClick={() => dispatch({ type: "increment" })}>Increment</
button>
      <button onClick={() => dispatch({ type: "reset" })}>Reset</button>
    </div>
  );
}
```

É importante ter em mente que funções reducer devem ser puras, ou seja, não podem ter efeitos colaterais. Além disso, o estado deve ser tratado como um objeto imutável, devemos apenas retornar o próximo estado sem alterar o atual.

Capítulo 12. Hooks customizados

Além de usar os *hooks* oferecidos pelo React, podemos definir nossos próprios *hooks*. Para isso, basta definirmos uma função seguindo a nomenclatura `useXXX`, onde XXX é o nome que desejar.

Hooks customizados podem chamar um ou mais hooks, o que nos permite extrair lógica dos componentes em hooks e reutilizarmos onde for necessário.

É importante seguir as mesmas regras de hooks que usamos em componentes, ou seja: (i) não chamar hooks dentro de loops ou comandos condicionais e (ii) chamar hooks apenas dentro de componentes ou outros hooks.

Capítulo 13. Class Components

Antes da versão 16.8, a única forma de criar componentes React contendo estado era por meio de classes JavaScript. Esses componentes são conhecidos como Class Components. A seguir, temos um exemplo minimalista de um Class Component.

```
class Counter extends React.Component<{}, { counter: number }> {
  constructor(props: {}) {
    super(props);
    this.state = { counter: 0 };
  }
  increment() {
    this.setState({ counter: this.state.counter + 1 });
  }
  render() {
    return (
      <div>
        <div>{this.state.counter}</div>
        <button onClick={() => this.increment()}>Increment</button>
      </div>
    );
  }
}
```

É importante observar que:

- Devemos estender React.Component.
- No construtor, recebemos props como parâmetro e devemos chamar o construtor da superclasse passando props.
- O estado é armazenado no atributo state e deve ser alterado sempre chamando-se o método setState.
- A renderização do componente é feita pelo método render.

13.1. Ciclo de vida do componente

Class Components permitem interagir com o ciclo de vida do componente ao sobrescrever métodos específicos. Dentre eles, destacamos os mais comuns a seguir:

- **`componentDidMount()`**: chamado uma vez, após a primeira renderização do componente.
- **`componentWillUnmount()`**: chamado uma vez, antes do componente ser destruído.
- **`componentDidUpdate(prevProps, prevState)`**: chamado toda vez que o componente é atualizado, ou seja, quando há mudanças no props ou no state. Esse método recebe como parâmetro os valores antigos de props e state.

Capítulo 14. Redux

A biblioteca Redux tem como objetivo ajudar na gestão do estado da aplicação, especialmente aquelas que possuem transições de estado complexa e que precisem de maior controle.

Mais do que apenas uma biblioteca, o Redux estabelece uma arquitetura para a aplicação, que pode ser resumida em três pontos principais:

1. O estado global da aplicação é centralizado no store. Existe uma única fonte da verdade.
2. O estado é imutável. A única forma de alterá-lo é emitindo uma action, um objeto que representa uma ação.
3. Dado o estado atual e uma action, um novo estado é calculado pela função reducer. Tal função não possui nenhum tipo de efeito colateral.

Uma aplicação Redux, conceitualmente, é como usarmos o hook useReducer para gerir todo o estado global da aplicação e disponibilizar o estado por meio de contexto para todos os componentes interessados.

14.1. Instalação do Redux

O Redux pode ser instalado via npm, com o seguinte comando:

```
npm install --save @reduxjs/toolkit react-redux
```

14.2. Definição do reducer e store

Para utilizar o Redux, precisamos, primeiro, definir um *reducer* e um *store*. Por meio do Redux *toolkit*, criamos um *reducer* com a função createSlice, conforme o

exemplo a seguir. Esta função recebe como parâmetro um nome, um valor inicial e um objeto reducers onde cada chave corresponde a uma action, que define uma função responsável pela mesma. Esta função recebe os objetos state e action como parâmetro.

```
const slice = createSlice({
  name: "counter",
  initialState: 0,
  reducers: {
    increment: (state) => {
      return state + 1;
    },
    reset: () => {
      return 0;
    },
  },
});

export const store = configureStore({
  reducer: {
    counter: slice.reducer,
  },
});
```

Após criar um slice com createSlice, podemos criar o store com configureStore.

14.3. Uso nos componentes React

Nos componentes React, usamos o estado e a função dispatch por meio dos hooks **useSelector** e **useDispatch**. No entanto, para uso em TypeScript, é necessário reexportar esses hooks com a tipagem correta, específica para sua aplicação, o que pode ser feito com o código abaixo.

```
export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
export const useAppDispatch = () => useDispatch<AppDispatch>();
export const useAppSelector: TypedUseSelectorHook<RootState> = useSelecto
r;
```

Feito isso, precisamos, ainda, adicionar o componente Provider do Redux, englobando os demais componentes da aplicação. O componente Provider recebe, como prop, o store, e internamente define um *provider* que permite o funcionamento dos *hooks* do Redux.

```
function App() {
  return <Provider store={store}>
    {/* componentes da aplicação*/}
  </Provider>
}
```

Finalmente, podemos usar **useAppSelector** para obter o estado da aplicação, e o hook **useAppDispatch** para disparar *actions*.

```
function MyComponent() {
  const counter = useAppSelector((state) => state.counter);
  const dispatch = useAppDispatch();
  return (
    <div>
      <div>{counter}</div>
      <button onClick={() => slice.actions.increment()}>Increment</button
>
    </div>
  );
}
```

Referências

ABRAMOV, Dan. *Documentação oficial do Redux*, 2021. Disponível em: https://redux.js.org/. Acesso em: 26 jan. 2022.

CSS Snapshot 2021. *W3C Group Note*, 31 dez. 2021. Disponível em: https://www.w3.org/TR/CSS/. Acesso em: 26 jan. 2022.

ECMASCRIPT 2020 language specification, 11th edition. *ecma international*, 2020. Disponível em: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/. Acesso em: 11 nov. 2021.

HTML Standard. *W3C,* 2021. Disponível em: https://www.w3.org/html/. Acesso em: 11 nov. 2021.

INTRODUCTION. *Google: Material Design*, 2021. Disponível em https://material.io/design/introduction. Acesso em: 26 jan. 2022.

MATERIAL-UI. Página oficial da biblioteca Material-UI, 2021. Disponível em: https://material-ui.com/. Acesso em: 26 jan. 2022.

MDN Web Docs. *Mozilla*, c2005-2022. Disponível em: https://developer.mozilla.org/. Acesso em: 26 jan. 2022.

QUICK Start. *React Training/React Router*. Disponível em: https://reactrouter.com/web/guides/quick-start. Acesso em: 26 jan. 2022.

TYPESCRIPT Documentation. *Microsoft: TypeScript*, c2012-2022. Disponível em: https://www.typescriptlang.org/docs. Acesso em: 26 jan. 2022.