# Como criar um site de comércio eletrônico com react

**Por** Deven Rathore

Javascript

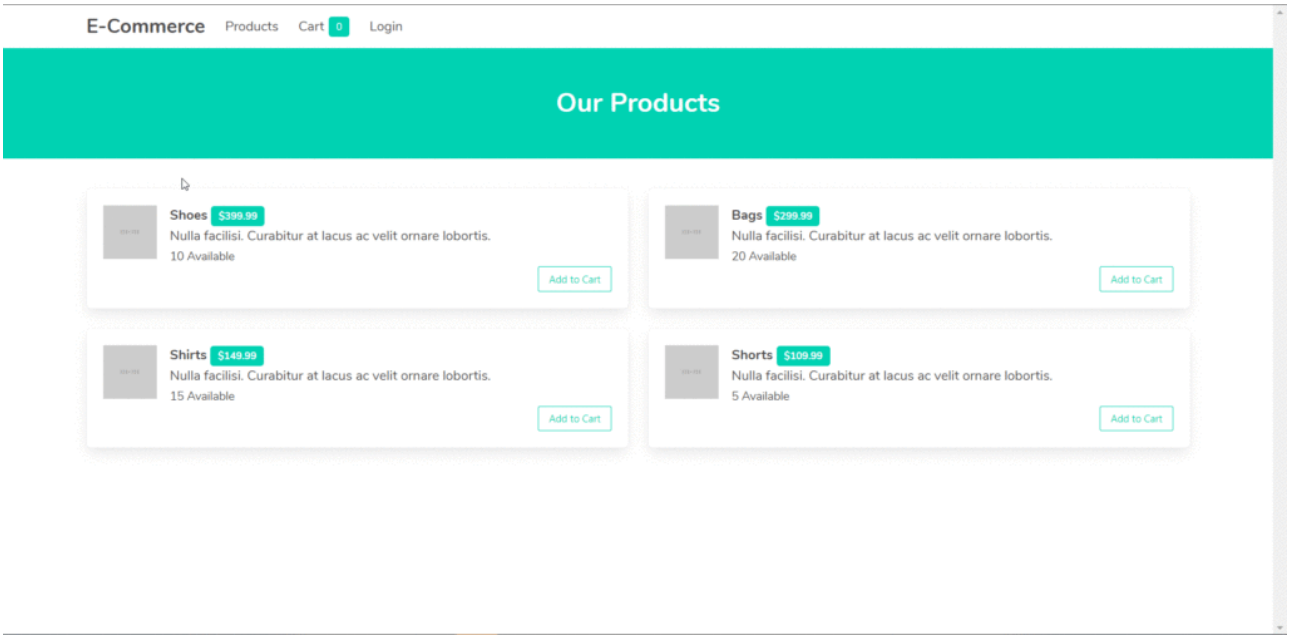
**14 de outubro de 2020**

**Compartilhar:**

**Neste tutorial, vamos ver como construir um aplicativo web de comércio eletrônico muito simples com o React. Não será o próximo Shopify, mas espero que sirva como uma maneira divertida de demonstrar o quão bem adaptado o React é para construir interfaces de usuário dinâmicas e envolventes.**

O aplicativo demonstrará um sistema básico de gerenciamento de carrinhos, bem como um método simples de lidar com a autenticação do usuário. Faremos uso do React Context como uma alternativa às estruturas de gestão do estado, como Redux ou MobX, e criaremos um back-end falso usando o pacote json-server.

Abaixo está uma captura de tela do que vamos construir:




O código deste aplicativo está disponível no GitHub.

## Pré-requisitos

Este tutorial assume que você tem um conhecimento básico de JavaScript e React. Se você é novo em Reagir, você pode gostar de conferir nosso guia de iniciantes.

Para criar o aplicativo, você precisará de uma versão recente do Node instalado em seu PC. Se este não for o caso, então vá até a página inicial do Node e baixe os binários corretos para o seu sistema. Alternativamente, você pode considerar usar um gerenciador de versão para instalar o Node. Temos um tutorial sobre o uso de um gerenciador de versões aqui.

O nó vem empacotado com npm, um gerenciador de pacotes para JavaScript, com o qual vamos instalar algumas das bibliotecas que usaremos. Você pode aprender mais sobre como usar npm aqui.

Você pode verificar se ambos estão instalados corretamente emitindo os seguintes comandos da linha de comando:

```
node -v
> 12.18.4

npm -v
> 6.14.8
```

Com isso feito, vamos começar criando um novo projeto React com a ferramenta Criar Aplicativos de Reação. Você pode instalar isso globalmente ou usar, assim: `npx`

```
npx create-react-app e-commerce
```

Quando isso terminar, mude para o diretório recém-criado:

```
cd e-commerce
```

Neste aplicativo, usaremos o React Router para lidar com o roteamento. Para instalar este módulo, execute:

```
npm install react-router-dom
```

Também precisaremos que json-server e json-server-auth criem nosso back-end falso para lidar com a autenticação:

```
npm install json-server json-server-auth
```

Vamos precisar de axios para fazer pedidos do Ajax para o nosso back-end falso.

```
npm install axios
```

E precisaremos de jwt-decode para que possamos analisar o JWT que nosso back-end responderá com:

```
npm install jwt-decode
```

Finalmente, usaremos a estrutura Bulma CSS para estilizar este aplicativo. Para instalar isso, execute o seguinte comando:

```
npm install bulma
```

# Começando

Primeiro, precisamos adicionar a folha de estilo à nossa aplicação. Para conseguir isso, adicionaremos uma declaração de importação para incluir este arquivo no arquivo da pasta. Isso aplicará a folha de estilo em todos os componentes do aplicativo: `index.js` `src`

```
import "bulma/css/bulma.css";
```

## Configuração de contexto

Como mencionado anteriormente, usaremos o React Context em todo o nosso aplicativo. Esta é uma adição relativamente nova ao React e fornece uma maneira de passar dados através da árvore componente sem ter que passar adereços para baixo manualmente em todos os níveis.

Se você quiser uma atualização sobre o uso do contexto em um aplicativo React, confira nosso tutorial "Como Substituir Redux com Ganchos de Reação e a API de contexto".

Em aplicações complexas onde a necessidade de contexto é geralmente necessária, pode haver múltiplos contextos, com cada um tendo seus próprios dados e métodos relacionados ao conjunto de componentes que requer os dados e métodos. Por exemplo, pode haver um para lidar com os componentes que usam dados relacionados ao produto e outro para o manuseio de dados relacionados à autenticação e dados do usuário. No entanto, para manter as coisas o mais simples possível, usaremos apenas uma instância de contexto. `ProductContext` `ProfileContext`

Para criar o contexto, criamos um arquivo e um arquivo no diretório do nosso aplicativo: `Context.js` `withContext.js` `src`

```
cd src
touch Context.js withContext.js
```

Em seguida, adicione o seguinte a: `Context.js`

```
import React from "react";
const Context = React.createContext({});
export default Context;
```

Isso cria o contexto e inicializa os dados do contexto para um objeto vazio. Em seguida, precisamos criar um invólucro de componentes, que usaremos para embrulhar componentes que usam os dados e métodos de contexto:

```
// src/withContext.js

import React from "react";
import Context from "./Context";

const withContext = WrappedComponent => {
  const WithHOC = props => {
    return (
      <Context.Consumer>
        {context => <WrappedComponent {...props} context={context} />}
      </Context.Consumer>
    );
  };

  return WithHOC;
};

export default withContext;
```

Isso pode parecer um pouco complicado, mas essencialmente tudo o que ele faz é fazer um componente de ordem superior,que anexa nosso contexto aos adereços de um componente embrulhado.

Quebrando-o um pouco, podemos ver que a função tem um componente React como seu parâmetro. Em seguida, ele retorna uma função que toma os adereços do componente como parâmetro. Dentro da função retornada, estamos embrulhando o componente em nosso contexto, em seguida, atribuindo-o o contexto como um suporte: . O bit garante que o componente retenha quaisquer adereços que foram passados para ele em primeiro lugar. `withContext` `context={context}` `{...props}`

```
import React from "react";
import withContext from "../withContext";

const Cart = props => {
  // We can now access Context as props.context
};

export default withContext(Cart);
```

## Andaimes fora do aplicativo

Agora, vamos criar uma versão esqueleto dos componentes que precisaremos para que a navegação básica do nosso aplicativo funcione corretamente. Estes são , e , e nós vamos colocá-los em um diretório dentro do diretório: `AddProducts` `Cart` `Login` `ProductList` `components` `src`

```
mkdir components
cd components
touch AddProduct.js Cart.js Login.js ProductList.js
```

Em adicionar: `AddProduct.js`

```
import React from "react";

export default function AddProduct() {
  return <>AddProduct</>
}
```

Em adicionar: `Cart.js`

```
import React from "react";

export default function Cart() {
```

```
  return <>Cart</>
}
```

Em adicionar: `Login.js`

```
import React from "react";

export default function Login() {
  return <>Login</>
}
```

E finalmente, em complemento: `ProductList.js`

```
import React from "react";

export default function ProductList() {
  return <>ProductList</>
}
```

Em seguida, precisamos configurar o arquivo. Aqui, vamos lidar com a navegação do aplicativo, bem como definir seus dados e métodos para gerenciá-lo. `App.js`

Primeiro, vamos configurar a navegação. Alterar da seguinte forma: `App.js`

```
import React, { Component } from "react";
import { Switch, Route, Link, BrowserRouter as Router } from "react-rc

import AddProduct from './components/AddProduct';
import Cart from './components/Cart';
import Login from './components/Login';
import ProductList from './components/ProductList';

import Context from "./Context";

export default class App extends Component {
```

```
    this.state = {
      user: null,
      cart: {},
      products: []
    };
    this.routerRef = React.createRef();
  }

  render() {
    return (
      <Context.Provider
        value={{
          ...this.state,
          removeFromCart: this.removeFromCart,
          addToCart: this.addToCart,
          login: this.login,
          addProduct: this.addProduct,
          clearCart: this.clearCart,
          checkout: this.checkout
        }}
      >
        <Router ref={this.routerRef}>
        <div className="App">
          <nav
            className="navbar container"
            role="navigation"
            aria-label="main navigation"
          >
            <div className="navbar-brand">
              <b className="navbar-item is-size-4 ">ecommerce</b>
              <label
                role="button"
                class="navbar-burger burger"
                aria-label="menu"
                aria-expanded="false"
                data-target="navbarBasicExample"
                onClick={e => {
                  e.preventDefault();
                  this.setState({ showMenu: !this.state.showMenu });
                }}
              >
                <span aria-hidden="true"></span>
                <span aria-hidden="true"></span>
                <span aria-hidden="true"></span>
```

```
            </div>
              <div className={`navbar-menu ${
                  this.state.showMenu ? "is-active" : ""
                }`}>
                <Link to="/products" className="navbar-item">
                  Products
                </Link>
                {this.state.user && this.state.user.accessLevel < 1 &&
                  <Link to="/add-product" className="navbar-item">
                    Add Product
                  </Link>
                )}
                <Link to="/cart" className="navbar-item">
                  Cart
                  <span
                    className="tag is-primary"
                    style={{ marginLeft: "5px" }}
                  >
                    { Object.keys(this.state.cart).length }
                  </span>
                </Link>
                {!this.state.user ? (
                  <Link to="/login" className="navbar-item">
                    Login
                  </Link>
                ) : (
                  <Link to="/" onClick={this.logout} className="navbar
                    Logout
                  </Link>
                )}
              </div>
            </nav>
            <Switch>
              <Route exact path="/" component={ProductList} />
              <Route exact path="/login" component={Login} />
              <Route exact path="/cart" component={Cart} />
              <Route exact path="/add-product" component={AddProduct}
              <Route exact path="/products" component={ProductList} />
            </Switch>
          </div>
        </Router>
      </Context.Provider>
    );
  }
```

Nosso componente será responsável pela inicialização dos dados do aplicativo e também definirá métodos para manipular esses dados. Primeiro, definimos os dados de contexto e os métodos que utilizam o componente. Os dados e métodos são passados como propriedade, no componente para substituir o objeto dado na criação do contexto. (Observe que o valor pode ser de qualquer tipo de dados.) Passamos o valor do Estado e alguns métodos, que definiremos em breve. `App` `Context.Provider` `value` `Provider`

Em seguida, construímos nossa navegação por aplicativos. Para isso, precisamos envolver nosso aplicativo com um componente, que pode ser (como no nosso caso) ou . Em seguida, definimos as rotas do nosso aplicativo usando os componentes. Também criamos o menu de navegação do aplicativo, com cada link usando o componente fornecido no módulo React Router. Adicionamos também uma referência, ao componente para nos permitir acessar o roteador de dentro do componente. `Router` `BrowserRouter` `HashRouter` `Switch` `Route` `Link` `rout`

Para testar isso, vá até a raiz do projeto (por exemplo) e inicie o servidor dev Create React App usando . Uma vez inicializado, seu navegador padrão deve abrir e você deve ver o esqueleto do nosso aplicativo. Certifique-se de clicar ao redor e certificar-se de que toda a navegação funciona. `/files/jim/Desktop/e-commerce` `npm start`

## Girando um back-end falso

Na próxima etapa, vamos configurar um back-end falso para armazenar nossos produtos e lidar com a autenticação do usuário. Como mencionado, para isso usaremos o json-server para criar uma API REST falsa e json-server-auth para adicionar um fluxo de autenticação simples baseado em JWT ao nosso

A maneira como o json-server funciona é que ele lê em um arquivo JSON do sistema de arquivos e usa isso para criar um banco de dados na memória com os pontos finais correspondentes para interagir com ele. Vamos criar o arquivo JSON agora. Na rota do seu projeto, crie uma nova pasta e nessa pasta crie um novo arquivo: `backend` `db.json`

```
mkdir backend
cd backend
touch db.json
```

Abra e adicione o seguinte conteúdo: `db.json`

```
{
  "users": [
    {
      "email": "regular@example.com",
      "password": "$2a$10$2myKMolZJoH.q.cyXClQXufY1Mc7ETKdSaQQCC6Fgtbe
      "id": 1
    },
    {
      "email": "admin@example.com",
      "password": "$2a$10$w8qB40MdYkMs3dgGGf0Pu.xxVOOzWdZ5/Nrkleo3Gqc8
      "id": 2
    }
  ],
  "products": [
    {
      "id": "hdmdu0t80yjkfqselfc",
      "name": "shoes",
      "stock": 10,
      "price": 399.99,
      "shortDesc": "Nulla facilisi. Curabitur at lacus ac velit ornare
      "description": "Cras sagittis. Praesent nec nisl a purus blandit
    },
    {
      "id": "3dc7fiyzlfmkfqseqam",
      "name": "bags",
      "stock": 20,
```

```
      "description": "Cras sagittis. Praesent nec nisl a purus blandit
    },
    {
      "id": "aoe8wvdxvrkfqsew67",
      "name": "shirts",
      "stock": 15,
      "price": 149.99,
      "shortDesc": "Nulla facilisi. Curabitur at lacus ac velit ornare
      "description": "Cras sagittis. Praesent nec nisl a purus blandit
    },
    {
      "id": "bmfrurdkswtkfqsf15j",
      "name": "shorts",
      "stock": 5,
      "price": 109.99,
      "shortDesc": "Nulla facilisi. Curabitur at lacus ac velit ornare
      "description": "Cras sagittis. Praesent nec nisl a purus blandit
    }
  ]
}
```

Estamos criando dois recursos aqui — e . Olhando para o recurso, você notará que cada usuário tem um ID, um endereço de e-mail e uma senha. A senha aparece como uma mistura de letras e números, pois é criptografada usando bcryptjs. É importante que você não armazene senhas em texto simples em *qualquer lugar* do seu aplicativo. `users` `products` `users`

Dito isto, a versão simples de texto de cada senha é simplesmente "senha" — sem aspas.

Agora inicie o servidor emitindo o seguinte comando a partir da raiz do projeto:

```
./node_modules/.bin/json-server-auth ./backend/db.json --port 3001
```

Isso iniciará o json-server em . Graças ao middleware json-server-auth, o recurso também nos dará um ponto final que podemos usar para simular o login no

aplicativo. `http://localhost:3001` `users` `/login`

Vamos experimentá-lo usando https://hoppscotch.io. Abra esse link em uma nova janela e mude o método para e a URL para . Em seguida, certifique-se de que o switch *de entrada Raw* está definido para *ligar* e digitar o seguinte como o Corpo de *Solicitação Bruta*: `POST` `http://localhost:3001/login`

```
{
  "email": "regular@example.com",
  "password": "password"
}
```

Clique em *Enviar* e você deve receber uma resposta (mais abaixo na página) que se pareça com isso:

```
{
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InJ
}
```

Esse é um JSON Web Token, que é válido por uma hora. Em um aplicativo normal com um back-end adequado, você salvaria isso no cliente e, em seguida, enviá-lo para o servidor sempre que você solicitasse um recurso protegido. O servidor validaria o token que recebeu e, se tudo fosse verificado, responderia com os dados solicitados.

Este ponto vale a pena repetir. Você precisa validar qualquer solicitação de um recurso protegido em seu servidor. Isso porque o código que é executado no cliente pode potencialmente ser projetado e adulterado.

Aqui está um link para a solicitação final no Hoppscotch. Você só precisa pressionar *Enviar*.

Se você quiser saber mais sobre o uso do JSON Web Tokens com Node.js, consulte nosso tutorial.

## Implementando autenticação no aplicativo React

Para esta seção vamos precisar dos pacotes de axios e jwt_decode em nosso aplicativo. Adicione as importações ao topo do arquivo: `App.js`

```
import axios from 'axios';
import jwt_decode from 'jwt-decode';
```

Se você der uma olhada no topo da classe, você verá que já estamos declarando um usuário no estado. Isso é inicialmente definido como nulo.

Em seguida, precisamos ter certeza de que o usuário está carregado quando o aplicativo é iniciado definindo o usuário na montagem do componente, como mostrado abaixo. Adicione este método ao componente, que carrega a última sessão do usuário do armazenamento local para o estado se ele existir: `App`

```
componentDidMount() {
  let user = localStorage.getItem("user");
  user = user ? JSON.parse(user) : null;
  this.setState({ user });
}
```

Em seguida, definimos os métodos e, que estão ligados ao contexto: `login` `logout`

```
login = async (email, password) => {
  const res = await axios.post(
    'http://localhost:3001/login',
    { email, password },
```

```
    return { status: 401, message: 'Unauthorized' }
  })

  if(res.status === 200) {
    const { email } = jwt_decode(res.data.accessToken)
    const user = {
      email,
      token: res.data.accessToken,
      accessLevel: email === 'admin@example.com' ? 0 : 1
    }

    this.setState({ user });
    localStorage.setItem("user", JSON.stringify(user));
    return true;
  } else {
    return false;
  }
}

logout = e => {
  e.preventDefault();
  this.setState({ user: null });
  localStorage.removeItem("user");
};
```

O método faz uma solicitação do Ajax ao nosso ponto final, passando-o seja lá o que o usuário inseriu no formulário de login (que faremos em um minuto). Se a resposta do ponto final tiver um código de status de 200, podemos assumir que as credenciais do usuário estavam corretas. Em seguida, decodificamos o token enviado na resposta do servidor para obter o e-mail do usuário, antes de salvar o e-mail, o token e o nível de acesso do usuário no estado. Se tudo correu bem, o método retorna, caso contrário. Podemos usar esse valor em nosso componente para decidir o que exibir. `login` `/login` `true` `false` `Login`

Observe que a verificação para o nível de acesso é muito superficial aqui e que não seria difícil para um usuário conectado e regular fazer-se um administrador. No entanto, supondo que as solicitações de recursos protegidos sejam validadas no servidor antes que uma resposta seja enviada, o usuário não seria capaz de

fazer muito mais do que ver um botão extra. A validação do servidor garantiria que eles não seriam capazes de obter quaisquer dados protegidos.

Se você quiser implementar uma solução mais robusta, você poderia fazer uma segunda solicitação para obter as permissões do usuário atual quando um usuário faz login ou sempre que o aplicativo é carregado. Isso infelizmente está fora do escopo deste tutorial.

O método libera o usuário do armazenamento estadual e local. `logout`

## Criando o componente de login

Em seguida, podemos lidar com o componente. Este componente faz uso dos dados de contexto. Para que ele tenha acesso a esses dados e métodos, ele tem que ser embrulhado usando o método que criamos anteriormente. `Login` `withContext`

Altere assim: `src/Login.js`

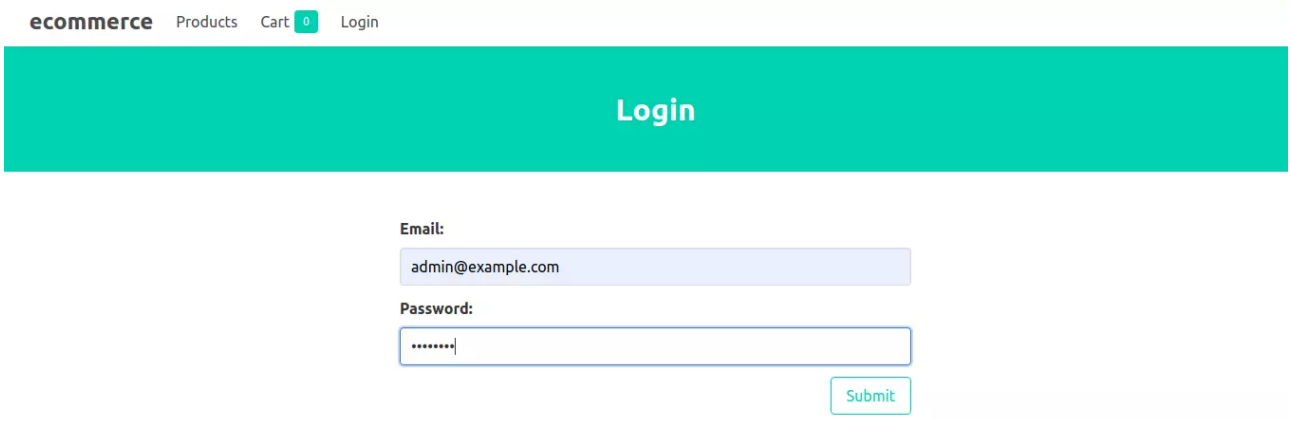
```
import React, { Component } from "react";
import { Redirect } from "react-router-dom";
import withContext from "../withContext";

class Login extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: "",
      password: ""
    };
  }

  handleChange = e => this.setState({ [e.target.name]: e.target.value,

  login = (e) => {
    e.preventDefault();
```

```
    if (!username || !password) {
      return this.setState({ error: "Fill all fields!" });
    }
    this.props.context.login(username, password)
      .then((loggedIn) => {
        if (!loggedIn) {
          this.setState({ error: "Invalid Credentails" });
        }
      })
  };

  render() {
    return !this.props.context.user ? (
      <>
        <div className="hero is-primary ">
          <div className="hero-body container">
            <h4 className="title">Login</h4>
          </div>
        </div>
        <br />
        <br />
        <form onSubmit={this.login}>
          <div className="columns is-mobile is-centered">
            <div className="column is-one-third">
              <div className="field">
                <label className="label">Email: </label>
                <input
                  className="input"
                  type="email"
                  name="username"
                  onChange={this.handleChange}
                />
              </div>
              <div className="field">
                <label className="label">Password: </label>
                <input
                  className="input"
                  type="password"
                  name="password"
                  onChange={this.handleChange}
                />
              </div>
              {this.state.error && (
                <div className="has-text-danger">{this.state.error}</d
```

```
              <div className="field is-clearfix">
                <button
                  className="button is-primary is-outlined is-pulled-r
                >
                  Submit
                </button>
              </div>
            </div>
          </div>
        </form>
      </>
    ) : (
      <Redirect to="/products" />
    );
  }
}

export default withContext(Login);
```

Este componente torna um formulário com duas entradas para coletar as credenciais de login do usuário. Na submissão, o componente chama o método, que é passado pelo contexto. Este módulo também faz questão de redirecionar para a página de produtos se o usuário já estiver logado. `login`

Se você agora for para http://localhost:3000/login,você deve ser capaz de fazer login com qualquer um dos combos de nome/senha acima mencionados.



# Criando as visualizações do produto

Agora precisamos buscar alguns produtos do nosso back-end para exibir em nosso aplicativo. Podemos novamente fazer isso no suporte de componentes no componente, como fizemos para o usuário logado: `App`

```
async componentDidMount() {
  let user = localStorage.getItem("user");
  const products = await axios.get('http://localhost:3001/products');
  user = user ? JSON.parse(user) : null;
  this.setState({ user,  products: products.data });
}
```

No trecho de código acima, marcamos o gancho do ciclo de vida como sendo assincnc, o que significa que podemos fazer uma solicitação ao nosso ponto final, em seguida, esperar que os dados sejam devolvidos antes de colocá-lo em estado. `componentDidMount` `/products`

Em seguida, podemos criar a página de produtos, que também funcionará como a página de aterrissagem do aplicativo. Esta página fará uso de dois componentes. O primeiro é , que mostrará o corpo da página, e o outro é o componente para cada produto da lista. `ProductList.js` `ProductItem.js`

Altere o componente, como mostrado abaixo: `Productlist`

```
import React from "react";
import ProductItem from "./ProductItem";
import withContext from "../withContext";

const ProductList = props => {
  const { products } = props.context;

  return (
```

```
        <div className="hero is-primary">
          <div className="hero-body container">
            <h4 className="title">Our Products</h4>
          </div>
        </div>
        <br />
        <div className="container">
          <div className="column columns is-multiline">
            {products && products.length ? (
              products.map((product, index) => (
                <ProductItem
                  product={product}
                  key={index}
                  addToCart={props.context.addToCart}
                />
              ))
            ) : (
              <div className="column">
                <span className="title has-text-grey-light">
                  No products found!
                </span>
              </div>
            )}
          </div>
        </div>
      </>
    );
};

export default withContext(ProductList);
```

Como a lista depende do contexto dos dados, nós o envolvemos com a função também. Este componente torna os produtos usando o componente, que ainda estamos para criar. Também passa um método do contexto (que ainda estamos para definir) para o . Isso elimina a necessidade de trabalhar com contexto diretamente no componente. `withContext` `ProductItem` `addToCart` `ProductItem` `ProductI`

Agora vamos criar o componente: `ProductItem`

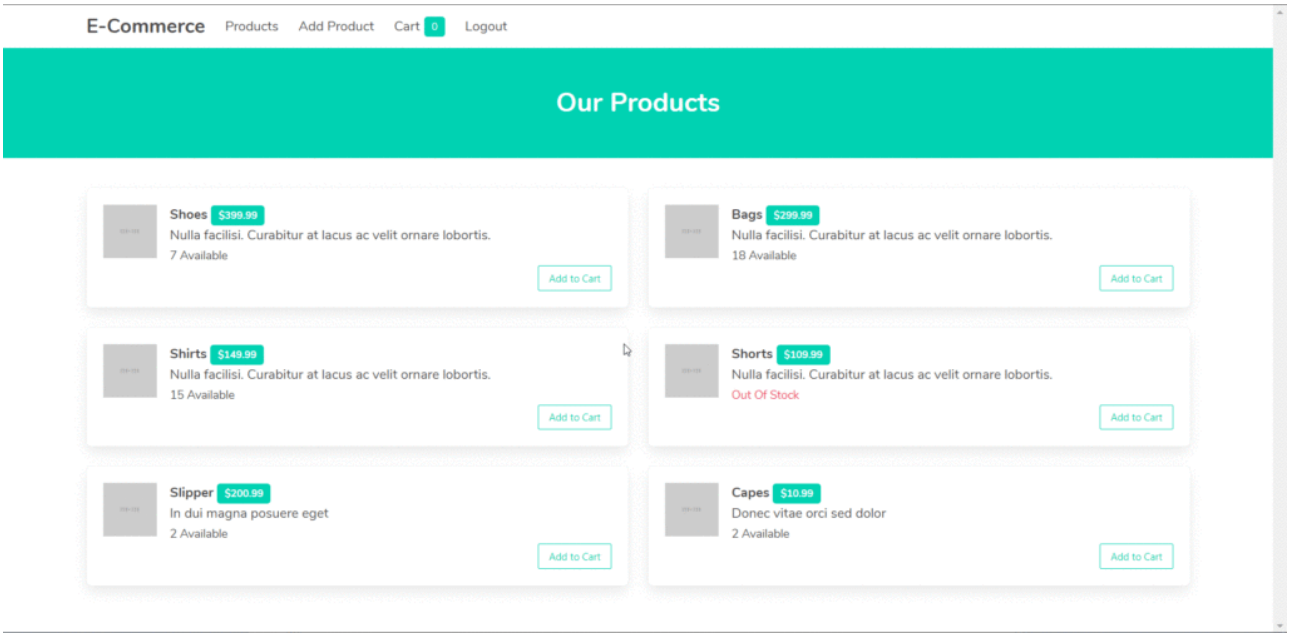
```
cd src/components
touch ProductItem.js
```

E adicione o seguinte conteúdo:

```
import React from "react";

const ProductItem = props => {
  const { product } = props;
  return (
    <div className=" column is-half">
      <div className="box">
        <div className="media">
          <div className="media-left">
            <figure className="image is-64x64">
              <img
                src="https://bulma.io/images/placeholders/128x128.png"
                alt={product.shortDesc}
              />
            </figure>
          </div>
          <div className="media-content">
            <b style={{ textTransform: "capitalize" }}>
              {product.name}{" "}
              <span className="tag is-primary">${product.price}</span>
            </b>
            <div>{product.shortDesc}</div>
            {product.stock > 0 ? (
              <small>{product.stock + " Available"}</small>
            ) : (
              <small className="has-text-danger">Out Of Stock</small>
            )}
            <div className="is-clearfix">
              <button
                className="button is-small is-outlined is-primary   is
                onClick={() =>
                  props.addToCart({
                    id: product.name,
                    product,
                    amount: 1
                  })
```

```
              >
                Add to Cart
              </button>
            </div>
          </div>
        </div>
      </div>
    );
};

export default ProductItem;
```

Este elemento exibe o produto em um cartão e também fornece um botão de ação para adicionar o produto ao carrinho do usuário.




## Adicionando um produto

Agora que temos algo para exibir em nossa loja, vamos criar uma interface para os usuários de administração adicionarem novos produtos. Primeiro, vamos definir o método para adicionar o produto. Faremos isso no componente, como mostrado abaixo: `App`

```
addProduct = (product, callback) => {
  let products = this.state.products.slice();
  products.push(product);
  this.setState({ products }, () => callback && callback());
};
```

Este método recebe o objeto e o anexa à matriz de produtos e, em seguida, salva-o para o estado do aplicativo. Ele também recebe uma função de retorno de chamada para executar na adição do produto com sucesso. `product`

Agora podemos continuar a preencher o componente: `AddProduct`

```
import React, { Component } from "react";
import withContext from "../withContext";
import { Redirect } from "react-router-dom";
import axios from 'axios';

const initState = {
  name: "",
  price: "",
  stock: "",
  shortDesc: "",
  description: ""
};

class AddProduct extends Component {
  constructor(props) {
    super(props);
    this.state = initState;
  }

  save = async (e) => {
    e.preventDefault();
    const { name, price, stock, shortDesc, description } = this.state;

    if (name && price) {
      const id = Math.random().toString(36).substring(2) + Date.now().

      await axios.post(
```

```
        { id, name, price, stock, shortDesc, description },
      )

      this.props.context.addProduct(
        {
          name,
          price,
          shortDesc,
          description,
          stock: stock || 0
        },
        () => this.setState(initState)
      );
      this.setState(
        { flash: { status: 'is-success', msg: 'Product created success
      );

    } else {
      this.setState(
        { flash: { status: 'is-danger', msg: 'Please enter name and pr
      );
    }
  };

  handleChange = e => this.setState({ [e.target.name]: e.target.value,

  render() {
    const { name, price, stock, shortDesc, description } = this.state;
    const { user } = this.props.context;

    return !(user && user.accessLevel < 1) ? (
      <Redirect to="/" />
    ) : (
      <>
        <div className="hero is-primary ">
          <div className="hero-body container">
            <h4 className="title">Add Product</h4>
          </div>
        </div>
        <br />
        <br />
        <form onSubmit={this.save}>
          <div className="columns is-mobile is-centered">
            <div className="column is-one-third">
```

```
          <label className="label">Product Name: </label>
          <input
            className="input"
            type="text"
            name="name"
            value={name}
            onChange={this.handleChange}
            required
          />
        </div>
        <div className="field">
          <label className="label">Price: </label>
          <input
            className="input"
            type="number"
            name="price"
            value={price}
            onChange={this.handleChange}
            required
          />
        </div>
        <div className="field">
          <label className="label">Available in Stock: </label>
          <input
            className="input"
            type="number"
            name="stock"
            value={stock}
            onChange={this.handleChange}
          />
        </div>
        <div className="field">
          <label className="label">Short Description: </label>
          <input
            className="input"
            type="text"
            name="shortDesc"
            value={shortDesc}
            onChange={this.handleChange}
          />
        </div>
        <div className="field">
          <label className="label">Description: </label>
          <textarea
```

```
                    type="text"
                    rows="2"
                    style={{ resize: "none" }}
                    name="description"
                    value={description}
                    onChange={this.handleChange}
                  />
                </div>
                {this.state.flash && (
                  <div className={`notification ${this.state.flash.statu
                    {this.state.flash.msg}
                  </div>
                )}
                <div className="field is-clearfix">
                  <button
                    className="button is-primary is-outlined is-pulled-r
                    type="submit"
                    onClick={this.save}
                  >
                    Submit
                  </button>
                </div>
              </div>
            </div>
          </form>
        </>
      );
    }
}

export default withContext(AddProduct);
```

Este componente faz uma série de coisas. Ele verifica se há um usuário atual armazenado em contexto e se esse usuário tem um de menos de 1 (isto é, se ele é um administrador). Se assim for, ele renderiza o formulário para adicionar um novo produto. Caso não, ele redireciona para a página principal do aplicativo. `accessLevel`

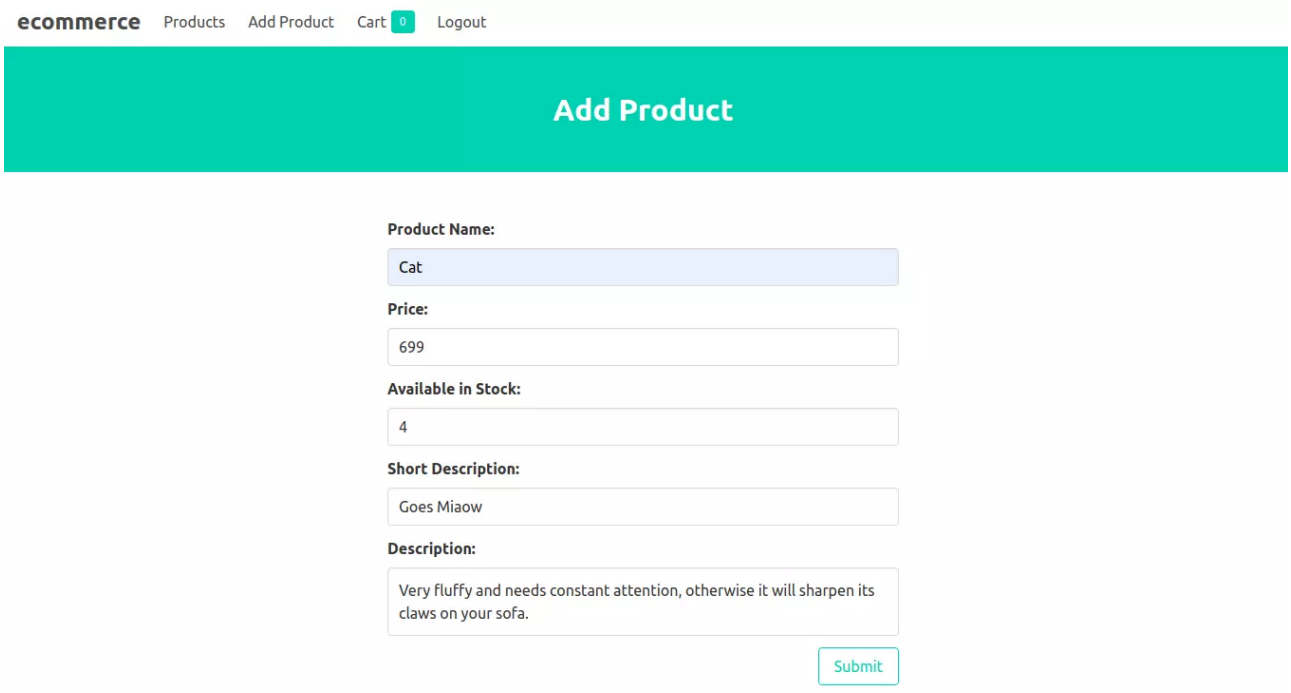
Mais uma vez, esteja ciente de que esta verificação pode ser facilmente ignorada

no servidor para garantir que o usuário possa criar novos produtos.

Supondo que o formulário seja prestado, existem vários campos para o usuário preencher (dos quais e são obrigatórios). O que quer que o usuário entre é rastreado no estado do componente. Quando o formulário é enviado, o método do componente é chamado, o que faz uma solicitação do Ajax ao nosso back-end para criar um novo produto. Também estamos criando um ID exclusivo (que json-server está esperando) e passando isso adiante, também. O código para isso veio de um thread no Stack Overflow. `name` `price` `save`

Finalmente, chamamos o método que recebemos via contexto, para adicionar o produto recém-criado ao nosso estado global e redefinir o formulário. Supondo que tudo isso tenha sido bem sucedido, estabelecemos uma propriedade em estado, que então atualizará a interface para informar ao usuário que o produto foi criado. `addProduct` `flash`

Se os campos estiverem faltando, definimos a propriedade para informar o usuário sobre isso. `name` `price` `flash`



Tire um segundo para verificar seu progresso. Faça login como administrador (e-mail: , senha: ) e certifique-se de que você veja um botão *Adicionar produto* na navegação. Navegue até esta página e use o formulário para criar alguns novos produtos. Finalmente, volte para a página principal e certifique-se de que os novos produtos estão aparecendo na lista de produtos. `admin@example.com` `password`

## Adicionando gerenciamento de carrinhos

Agora que podemos adicionar e exibir produtos, a última coisa a fazer é implementar o gerenciamento do nosso carrinho. Nós já iniciamos nosso carrinho como um objeto vazio, mas também precisamos ter certeza de que carregamos o carrinho existente do armazenamento local na carga dos componentes. `App.js`

Atualize o método da seguinte forma: `componentDidMount` `App.js`

```
async componentDidMount() {
  let user = localStorage.getItem("user");
  let cart = localStorage.getItem("cart");

  const products = await axios.get('http://localhost:3001/products');
  user = user ? JSON.parse(user) : null;
  cart = cart? JSON.parse(cart) : {};

  this.setState({ user,  products: products.data, cart });
}
```

Em seguida, precisamos definir as funções do carrinho (também em ). Primeiro, criaremos o método: `App.js` `addToCart`

```
addToCart = cartItem => {
  let cart = this.state.cart;
  if (cart[cartItem.id]) {
```

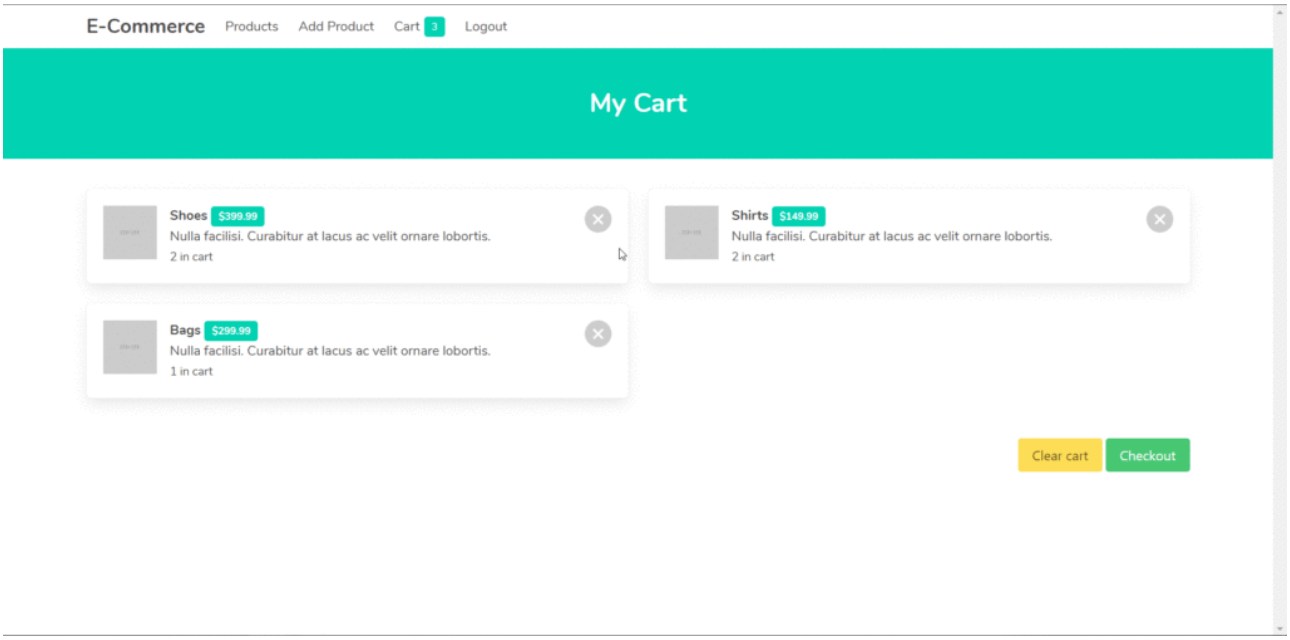
```
  } else {
    cart[cartItem.id] = cartItem;
  }
  if (cart[cartItem.id].amount > cart[cartItem.id].product.stock) {
    cart[cartItem.id].amount = cart[cartItem.id].product.stock;
  }
  localStorage.setItem("cart", JSON.stringify(cart));
  this.setState({ cart });
};
```

Este método anexa o item usando o ID do item como chave para o objeto do carrinho. Estamos usando um objeto em vez de uma matriz para o carrinho para permitir a recuperação fácil de dados. Este método verifica o objeto do carrinho para ver se existe um item com essa tecla. Se isso acontecer, aumenta a quantidade; caso contrário, cria uma nova entrada. A segunda instrução garante que o usuário não pode adicionar mais itens do que realmente estão disponíveis. O método então salva o carrinho para o estado, que é passado para outras partes da aplicação através do contexto. Finalmente, o método salva o carrinho atualizado para armazenamento local para persistência. `if`

Em seguida, definiremos o método para remover um produto específico do carrinho do usuário e remover todos os produtos do carrinho do usuário: `removeFromCart` `clearCart`

```
removeFromCart = cartItemId => {
  let cart = this.state.cart;
  delete cart[cartItemId];
  localStorage.setItem("cart", JSON.stringify(cart));
  this.setState({ cart });
};

clearCart = () => {
  let cart = {};
  localStorage.removeItem("cart");
  this.setState({ cart });
};
```

O método remove um produto usando a chave do produto fornecida. Em seguida, atualiza o estado do aplicativo e o armazenamento local de acordo. O método redefine o carrinho para um objeto vazio em estado e remove a entrada do carrinho no armazenamento local. `removeCart` `clearCart`




Agora, podemos proceder para fazer a interface de usuário do carrinho. Semelhante à lista de produtos, conseguimos isso usando dois elementos: o primeiro, que torna o layout da página, e uma lista de itens de carrinho usando o segundo componente: `Cart.js` `CartItem.js`

```
// ./src/components/Cart.js

import React from "react";
import withContext from "../withContext";
import CartItem from "./CartItem";

const Cart = props => {
  const { cart } = props.context;
  const cartKeys = Object.keys(cart || {});
  return (
    <>
      <div className="hero is-primary">
        <div className="hero-body container">
          <h4 className="title">My Cart</h4>
```

```
      </div>
      <br />
      <div className="container">
        {cartKeys.length ? (
          <div className="column columns is-multiline">
            {cartKeys.map(key => (
              <CartItem
                cartKey={key}
                key={key}
                cartItem={cart[key]}
                removeFromCart={props.context.removeFromCart}
              />
            ))}
            <div className="column is-12 is-clearfix">
              <br />
              <div className="is-pulled-right">
                <button
                  onClick={props.context.clearCart}
                  className="button is-warning "
                >
                  Clear cart
                </button>{" "}
                <button
                  className="button is-success"
                  onClick={props.context.checkout}
                >
                  Checkout
                </button>
              </div>
            </div>
          </div>
        ) : (
          <div className="column">
            <div className="title has-text-grey-light">No item in cart
          </div>
        )}
      </div>
    </>
  );
};

export default withContext(Cart);
```

O componente também passa um método do contexto para o . O componente gira através de uma matriz dos valores do objeto do carrinho de contexto e retorna um para cada um. Ele também fornece um botão para limpar o carrinho do usuário. `Cart` `CartItem` `Cart` `CartItem`

Em seguida é o componente, que é muito parecido com o componente, mas para algumas mudanças sutis: `CartItem` `ProductItem`

Vamos criar o componente primeiro:

```
cd src/components
touch CartItem.js
```

Em seguida, adicione o seguinte conteúdo:

```
import React from "react";

const CartItem = props => {
  const { cartItem, cartKey } = props;

  const { product, amount } = cartItem;
  return (
    <div className=" column is-half">
      <div className="box">
        <div className="media">
          <div className="media-left">
            <figure className="image is-64x64">
              <img
                src="https://bulma.io/images/placeholders/128x128.png"
                alt={product.shortDesc}
              />
            </figure>
          </div>
          <div className="media-content">
            <b style={{ textTransform: "capitalize" }}>
              {product.name}{" "}
              <span className="tag is-primary">${product.price}</span>
```

```
            <div>{product.shortDesc}</div>
            <small>{`${amount} in cart`}</small>
          </div>
          <div
            className="media-right"
            onClick={() => props.removeFromCart(cartKey)}
          >
            <span className="delete is-large"></span>
          </div>
        </div>
      </div>
    </div>
  );
};

export default CartItem;
```

Este componente mostra as informações do produto e o número de itens selecionados. Ele também fornece um botão para remover o produto do carrinho.

Finalmente, precisamos adicionar o método de checkout no componente: `App`

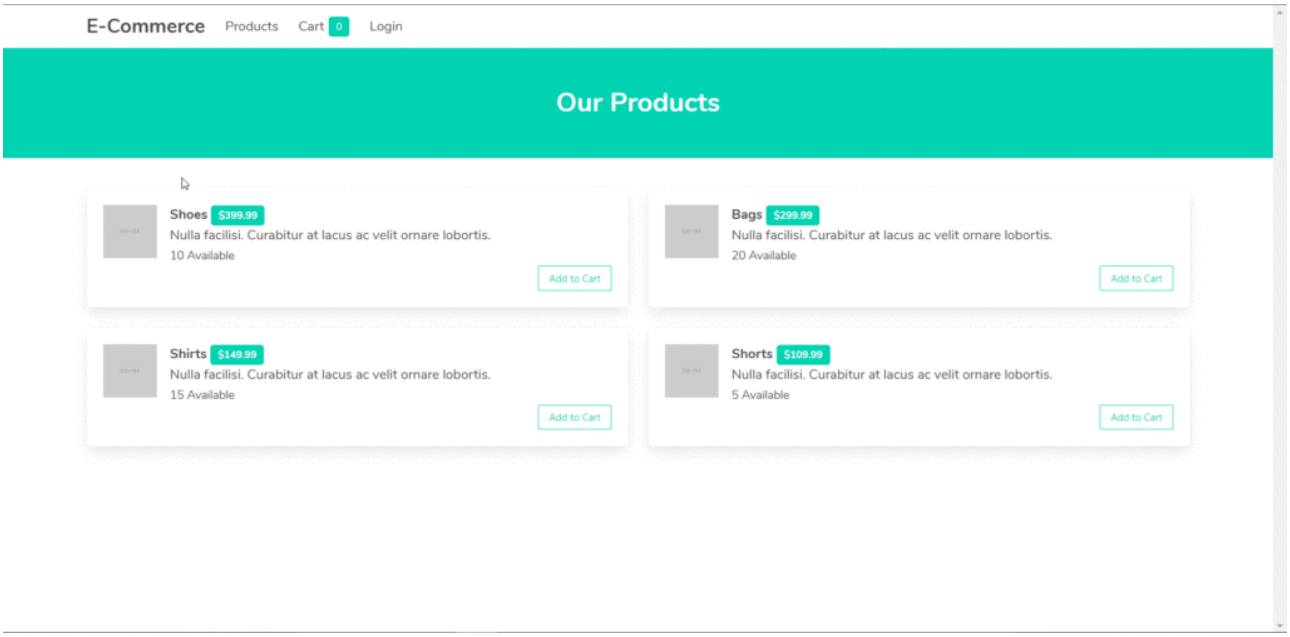
```
checkout = () => {
  if (!this.state.user) {
    this.routerRef.current.history.push("/login");
    return;
  }

  const cart = this.state.cart;

  const products = this.state.products.map(p => {
    if (cart[p.name]) {
      p.stock = p.stock - cart[p.name].amount;

      axios.put(
        `http://localhost:3001/products/${p.id}`,
        { ...p },
      )
    }
```

```
    this.setState({ products });
    this.clearCart();
  };
```

Este método verifica se um usuário está logado antes de prosseguir. Se o usuário não estiver logado, ele redirecionará o usuário para a página de login usando a referência do roteador que anexamos ao componente anteriormente. `Router`

Normalmente, em um site de comércio eletrônico regular, é aqui que o processo de faturamento ocorreria, mas para o nosso aplicativo, vamos apenas assumir que o usuário pagou e, portanto, remover seus itens comprados da lista de itens disponíveis. Também usaremos axios para atualizar o nível de estoque em nosso back-end.

Com isso, conseguimos completar nosso carrinho básico de compras.




## Conclusão

No decorrer deste tutorial, usamos o React para limpar a interface de um carrinho de compras básico. Usamos o contexto para mover dados e métodos entre vários

componentes e json-server para persistir os dados. Também usamos auth json-server para implementar um fluxo básico de autenticação.

Esta aplicação não é de forma alguma um produto acabado e poderia ser melhorada em muitos aspectos. Por exemplo, o próximo passo seria adicionar um back-end adequado com um banco de dados e realizar verificações de autenticação no servidor. Você também pode dar aos usuários de administração a capacidade de editar e excluir produtos.

Espero que tenham gostado deste tutorial. Por favor, não se esqueça que o código para este aplicativo está disponível no GitHub.

Quer mergulhar em mais react? Confira Padrões de Design de Reação e Práticas Recomendadas e muitos outros recursos react no SitePoint Premium.

**Deven Rathore**

A Deven é uma empreendedora e desenvolvedora full-stack, constantemente aprendendo e experimentando coisas novas. Atualmente, gerencia CodeSource.io e Dunebook.com e trabalha ativamente em projetos como o WrapPixel

# Adicione funcionalidade do Office ao seu aplicativo web com o OnlyOffice

Por Beardscript

Javascript

2 de dezembro de 2020

Compartilhar:

*Este artigo foi criado em parceria com o OnlyOffice. Obrigado por apoiar os parceiros que tornam o SitePoint possível.*

**Sempre que nos encontramos tentando adicionar qualquer funcionalidade complexa a um aplicativo, surge a pergunta: "Devo rolar a minha própria?" E a menos que seu objetivo seja construir essa funcionalidade, a resposta é quase sempre um "não" reto.**

O que você precisa é de algo para ajudá-lo a chegar a um MVP o mais rápido possível, e a melhor maneira de conseguir isso é usar uma solução completa fora da caixa que pode ajudá-lo a economizar tempo, o que, por sua vez, se traduz em economia nos custos de desenvolvimento.

Eu presumo que você ainda está aqui porque o acima ressoa com você. Então, agora que estamos em sincronia, o que eu quero mostrar neste artigo é como é fácil integrar o OnlyOffice em seu aplicativo web.

## O que é o Só Escritório?

> OnlyOffice oferece o pacote office mais rico em recursos disponível, altamente compatível com formatos de arquivos Microsoft Office e OpenDocument. Exibir, editar e trabalhar de forma colaborativa com documentos, planilhas e apresentações diretamente do seu aplicativo web.

Neste caso, vamos usar a Developer Edition,já que é a melhor para o nosso propósito, mas se você está procurando se integrar com outros serviços como o SharePoint, então você deve conferir a Edição de Integração.

The office suite has several editions. In this article we are going to use Developer Edition, because we want to integrate the editors into the app which will later be delivered to many users as a cloud service or on-premise installation.

If you want to use OnlyOffice within an existing sync & share solution, you should check out Enterprise Edition. A list of integrations is here.

### Developer Edition

The Developer Edition not only gives you enough freedom to integrate the editors within your app, but it also comes with a “White Label” option which lets you fully customize the editors to use them under your own brand.

## Document Server Integration

To integrate with your web app, you first need to download the OnlyOffice Docs (packaged as Document Server) and set it up on your local server.

After you’ve installed it you can start implementing the requests to handle documents on your server. OnlyOffice provides some very nice examples for .NET, Java, Node.js, PHP, Python and Ruby.

You can download the Document Server and your preferred example and try it straight away on your machine.

I'll demonstrate how you can go about starting to integrate into your app. For this purpose, we'll use a very simple example with Node.js and Express. I won't go into much detail on the implementation, I'll lay out the bare bone essentials and let you fill in the blanks to build a robust and scalable system.

I have an app with the following structure:

```
- node_modules
- public
    - backups
    - css
        - main.css
    - documents
        - sample.docx
    - javascript
        - main.js
    - samples
        - new.docx
        - new.xlsx
        - new.pptx
- app.js
- index.html
- package.json
```

We'll use the folder to store the documents. The file is where our Express app code is, and is where we'll show our documents. I've dropped a file in the documents folder for testing purposes. `public/documents` `app.js` `index.html` `sample.docx`

The tree files inside are the blank files that we'll copy when "creating" new files. `public/samples/`

The folder, as you'll see later, will not only help us keep backups of previous versions but also assist us in generating the unique identifier for our documents after modifying them. `backups`

The and files will be used by the . We'll look into that later. `public/css/main.css` `public/javascript/main.js` `index.html`

Let's take a look at the file: `app.js`

```
const express = require('express');
const bodyParser = require("body-parser");
const path = require('path');
const fs = require('fs');
const syncRequest = require('sync-request');

const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

app.use(express.static("public"));

app.get("/", (req, res) => {
  res.sendFile(path.join(__dirname, "/index.html"));
});

const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`App listening on http://localhost:
```

What we're doing is serving the files as . `localhost:3000/documents/filename`

I've also gotten ahead of myself and added , , and . These are not relevant right now but we'll use them later. `syncRequest` `fs` `bodyParser`

## Fetch Documents

To show the available documents we'll need to get a list of all the filenames and send them to the client. We'll create the route for this: `/documents`

```
app.get("/documents", (req, res) => {
  const docsPath = path.join(__dirname, "public/documents");
  const docsPaths = fs.readdirSync(docsPath);

  const fileNames = [];

  docsPaths.forEach(filePath => {
    const fileName = path.basename(filePath);
    fileNames.push(fileName);
  });

  res.send(fileNames);
});
```

## Create Documents

At the beginning we'll just have a sample document, but that's no fun at all. Let's add a route to assist us with adding some files. We'll simply take a and copy the corresponding template into the folder with its new name: `/create` `fileName` `public/documents`

```
app.post("/create", async (req, res) => {
  const ext = path.extname(req.query.fileName);
  const fileName = req.query.fileName;

  const samplePath = path.join(__dirname, "public/samples", "new" + ex
  const newFilePath = path.join(__dirname, "public/documents", fileNam

  // Copy the sample file to the documents folder with its new name.
  try {
    fs.copyFileSync(samplePath, newFilePath);
    res.sendStatus(200);
  } catch (e) {
```

```
  }
});
```

## Delete Documents

We also need a way to delete documents. Let's create a the route: `/delete`

```
app.delete("/delete", (req, res) => {
  const fileName = req.query.fileName;
  const filePath = path.join(__dirname, "public/documents", fileName);

  try {
    fs.unlinkSync(filePath);
    res.sendStatus(200);
  } catch (e) {
    res.sendStatus(400);
  }
});
```

This one's super simple. We'll delete the file and send a status code to let the user know it all went fine. Otherwise, they'll get a status code. `200` `400`

## Save Documents

So far, we can open our documents for editing, but we have no way of saving our changes. Let's do that now. We'll add a route to save our files: `/track`

```
app.post("/track", async (req, res) => {
  const fileName = req.query.fileName;

  const backupFile = filePath => {
    const time = new Date().getTime();
    const ext = path.extname(filePath);
    const backupFolder = path.join(__dirname, "public/backups", fileNa

    // Create the backups folder if it doesn't exist
```

```
    // Remove previous backup if any
    const previousBackup = fs.readdirSync(backupFolder)[0];
    previousBackup && fs.unlinkSync(path.join(backupFolder, previousBa

    const backupPath = path.join(backupFolder, time + ext);

    fs.copyFileSync(filePath, backupPath);
  }

  const updateFile = async (response, body, path) => {
    if (body.status == 2) {
      backupFile(path);
      const file = syncRequest("GET", body.url);
      fs.writeFileSync(path, file.getBody());
    }

    response.write("{\"error\":0}");
    response.end();
  }

  const readbody = (request, response, path) => {
    const content = "";
    request.on("data", function (data) {
      content += data;
    });
    request.on("end", function () {
      const body = JSON.parse(content);
      updateFile(response, body, path);
    });
  }

  if (req.body.hasOwnProperty("status")) {
    const filePath = path.join(__dirname, "public/documents", fileName
    updateFile(res, req.body, filePath);
  } else {
    readbody(req, res, filePath);
  }
});
```

This is a tricky one, since it's going to be used by the Document Server when the file is saved by the editor. As you can see, we're returning, which tells the server

that it's all good. `"{\"error\":0}"`

When the editor is closed, the current version of the file will be backed up in with the current time in milliseconds as the file's name. We'll use the file's name later in the front end, as you'll see. `public/backups/fileName-history/`

In this example, we're replacing the previous backup every time we save a new one. How would you go about keeping more backups?

## Fetching backups

We'll need a way to get the backups for a particular file, so we're adding a route to handle this: `/backups`

```
app.get("/backups", (req, res) => {
  const fileName = req.query.fileName;
  const backupsPath = path.join(__dirname, "public/backups", fileName

  if (!fs.existsSync(backupsPath)) {
    return res.send([]);
  }

  const backupsPaths = fs.readdirSync(backupsPath);

  const fileNames = [];

  backupsPaths.forEach(filePath => {
    const fileName = path.basename(filePath);
    fileNames.push(fileName);
  });

  res.send(fileNames);
});
```

Here we're making sure that the backup folder for that file exists, and returning an array of all the backup files in that folder. Yes, this will help you in your task of

# Opening a Document in the Browser

We'll see how we can go about opening our documents to edit directly in the browser using OnlyOffice Docs.

First, we'll create a simple HTML file:

```
<!DOCTYPE html>
<html>

<head>
  <title>OnlyOffice Example</title>

  <meta name="viewport" content="width=device-width, initial-scale=1.0
  <link rel="stylesheet" href="/public/css/main.css">
</head>

<body>
  <div id="placeholder"></div>
  <div id="documents">
```

```
    <div id="document-controls">
      <div onclick="createDocument('.docx')">Create docx</div>
      <div onclick="createDocument('.xlsx')">Create xlsx</div>
      <div onclick="createDocument('.pptx')">Create pptx</div>
    </div>
  </div>
  <script type="text/javascript" src="http://localhost:8080/web-apps/a
  <script type="text/javascript" src="/public/javascript/main.js"></sc
</body>

</html>
```

As you can see, there's not much to this file. We have the div where the editor will be attached. Then there's the div, which contains the controls to create documents and a container for the list of file names. `placeholder` `documents`

Below that, we have the script with the JavaScript API for the Document Server. Keep in mind that you might have to replace the host with the location of your Document Server. If you installed it with the command I gave you, you should be good to go. `Docker`

Last but not least, there's the tag, where we import our front-end JavaScript, and the file, where we'll have global access to the object. `script` `main.js` `DocsAPI`

## CSS

Before we get to coding, let's wrap up the layout with some CSS to make our app more usable and less ugly. Add the following to : `main.css`

```
html,
body {
  font-family: monospace;
  height: 100%;
  margin: 0;
  background-color: lavender;
  color: aliceblue;
```

```
h1 {
  color: lightslategray;
  display: inline-block;
}

#placeholder {
  height: 100%;
}

#documents {
  text-align: center;
}

#document-controls {
  text-align: center;
  margin: 5px;
}

#document-controls>div {
  display: inline-block;
  font-size: 15px;
  cursor: pointer;
  padding: 10px;
  background: mediumaquamarine;
}

#documents-list {
  padding: 5px;
  max-width: 400px;
  margin: auto;
}

.document {
  cursor: pointer;
  font-size: 20px;
  text-align: left;
  padding: 5px;
  margin: 2px;
  background-color: lightsteelblue;
}

.delete-doc {
  color: lightslategray;
```

```
  margin: 0 5px 0 5px;
}
```

## Showing available documents

With that out of the way, we're ready to start coding the front end. We'll start by listing the files in the folder. Go to the and add the following code: `documents` `main.js`

```
const params = new URLSearchParams(window.location.search);
const fileName = params.get("fileName");

if (fileName) {
  editDocument(fileName);
} else {
  listDocuments();
}

function listDocuments() {
  // Hide the editor placeholder
  document.getElementById("placeholder").style.display = "none";
  // Remove old list
  const oldList = document.getElementById("documents-list");
  oldList && oldList.remove();
  // Create new container
  const documentsHtml = document.getElementById("documents");
  const docsListHtml = document.createElement("div");
  docsListHtml.id = "documents-list";

  documentsHtml.appendChild(docsListHtml);

  const req = new XMLHttpRequest();

  req.addEventListener("load", function (evt) {
    const docs = JSON.parse(this.response);

    docs.forEach(doc => {
      addDocumentHtml(doc);
    });
  });
```

```
  req.send();
}

function addDocumentHtml(fileName) {
  const docsListHtml = document.getElementById("documents-list");

  const docElement = document.createElement("div");
  docElement.id = fileName;
  docElement.textContent = fileName;
  docElement.setAttribute("class", "document");

  docElement.onclick = () => {
    openDocument(fileName);
  }

  const deleteElement = document.createElement("span");
  deleteElement.textContent = "X";
  deleteElement.setAttribute("class", "delete-doc");

  deleteElement.onclick = evt => {
    evt.stopPropagation();
    evt.preventDefault();
    deleteDocument(fileName);
  }

  docElement.appendChild(deleteElement);
  docsListHtml.appendChild(docElement);
}

function openDocument(fileName) {
  const url = "/?fileName=" + fileName;
  open(url, "_blank");
}
```

Here at the top, we're getting the query parameters to find out if we're opening a file or not. If we are, we'll call the function. Don't worry, we'll create that one later. `editDocument`

If we're not opening a file, we want to show a list of the available files and the controls to create more. In , we first make sure that we hide the and clear up the

all the files, iterate through them, and create the corresponding elements. We'll identify each element with the filename as the ID. This way we can easily retrieve them later. `listDocuments` `placeholder` `/documents`

Notice that we're calling the function, which we'll reuse later to add new files. `addDocumentHtml`

For each of these documents, we're also calling the , which we defined at the bottom, and on the cross symbol we're calling the , which we'll define next. `openDocument` `deleteDocument`

## Deleting documents

To delete our documents, we'll prompt the user if they're sure before we go ahead and call the route and go nuclear on that file. Instead of wasting another call to our API, we're checking that the returned status is to delete the DOM elements directly: `/delete` `200`

```
function deleteDocument(fileName) {
  const canContinue = confirm("Are you sure you want to delete " + fil

  if (!canContinue) {
    return;
  }

  const req = new XMLHttpRequest();

  req.addEventListener("load", function (evt) {
    if (this.status === 200) {
      return removeDocumentHtml(fileName);
    }

    alert("Could not delete " + fileName);
  });

  req.open("DELETE", "/delete?fileName=" + fileName);
  req.send();
```

```
function removeDocumentHtml(fileName) {
  const el = document.getElementById(fileName);
  el && el.remove();
}
```

## Create documents

Remember that function we were calling in the of the document creation controls? Here you go: `onclick`

```
function createDocument(extension) {
  const name = prompt("What's the name of your new document?");
  const fileName = name + "." + extension;

  const req = new XMLHttpRequest();

  req.addEventListener("load", function (evt) {
    if (this.status === 200) {
      addDocumentHtml(fileName);
      return;
    }

    alert("Could not create " + fileName);
  });

  req.open("POST", "/create?fileName=" + fileName);
  req.send();
}
```

Very simple. We prompt the name, call the route with that as the parameter, and if the status comes back as we call the to add the DOM elements directly. `/create` `fileName` `200` `addDocumentHtml`

## Opening documents in OnlyOffice Docs

Now we need to define the function. Add the following code to

```
async function editDocument(fileName) {
  document.getElementById("documents").style.display = "none";

  const extension = fileName.substring(fileName.lastIndexOf(".") + 1);
  const documentType = getDocumentType(extension);
  const documentKey = await generateKey(fileName);

  console.log(documentKey);

  new DocsAPI.DocEditor("placeholder", {
    document: {
      fileType: extension,
      key: documentKey,
      title: fileName,
      url: "http://192.168.0.7:3000/documents/" + fileName,
    },
    documentType,
    editorConfig: {
      callbackUrl: "http://192.168.0.7:3000/track?fileName=" + fileNam
    },
    height: "100%",
    width: "100%",
  });
}

function generateKey(fileName) {
  return new Promise(resolve => {
    const req = new XMLHttpRequest();

    req.addEventListener("load", function (evt) {
      const backups = JSON.parse(this.response);
      const backupName = backups[0];
      const key = backupName ? backupName.substring(0, backupName.inde
      resolve(String(key));
    });

    req.open("GET", "/backups?fileName=" + fileName);
    req.send();
  });
}

function getDocumentType(extension) {
  const documentTypes = {
```

```
    spreadsheet: ["xls", "xlsx", "xlsm", "xlt", "xltx", "xltm", "ods",
    presentation: ["pps", "ppsx", "ppsm", "ppt", "pptx", "pptm", "pot'
  }

  if (documentTypes.text.indexOf(extension) >= 0) {
    return "text";
  }
  if (documentTypes.spreadsheet.indexOf(extension) >= 0) {
    return "spreadsheet";
  }
  if (documentTypes.presentation.indexOf(extension) >= 0) {
    return "presentation";
  }
}
```

So, we've added three functions. Let's focus on the last two first. (We'll talk about in a moment.) `editDocument`

The will also assist us by generating the key. This is a unique document identifier used for document recognition by the service. It can have a maximum length of 20 and no special characters. And here's the trick: it has to be regenerated every time the document is saved. Do you see where this is going? Exactly! We're going to profit from our backup file names to generate our keys. `generateKey`

As you can see, to generate the key we're retrieving our only backup (if any) and using its name or otherwise simple getting the current time in milliseconds if there are none.

What would have to change in that function if you were to support more backups? *[Runs away]*

The will return either , or . OnlyOffice needs this to know which editor to open. `getDocumentType` `text` `spreadsheet` `presentation`

The is what we're here for. This is what you've been waiting for all along. Here we instantiate the object passing the ID of our div and an object with a bunch of configurations. `editDocument` `DocEditor` `placeholder`

## DocEditor Configuration

What I've shown you so far are the minimum required options to instantiate the . You should check out the Advanced Parameters section in the docs to see how you can profit from all the different options. In the meantime, let me take you through the fundamentals. `DocEditor`

At the top, we have the document field which takes an object containing the information regarding the document that we want to open.

Then we have the , which, as we saw earlier, can be either , , or . `documentType` `text` `spreadsheet` `presentation`

Right below that is the editorConfig object, which lets you set things like , and , among other things. In this case, we're just using the , which is the URL to the route that the Document Server will use to save the file. `spellcheck` `unit` `zoom` `callbackUrl` `/track`

## Conclusion

We've reached the end, and you've hopefully learned how to set up and integrate OnlyOffice Docs with your web app. There's a lot we're leaving out, like permissions, sharing, customization and a lot of other things that you can do with OnlyOffice.

I hope you've got enough information to keep improving your product, or maybe even inspiration to start a new project from scratch. There's no time like the present.

Alright, I'll see you in the next one. In the meantime, keep coding and remember to have fun while you're at it!

**Beardscript**

Além de desenvolvedor de software, também sou massagista, músico entusiasmado e escritor de ficção hobbyist. Adoro viajar, assistir programas de TV de boa qualidade e, claro, jogar videogame.

## Livros Populares

Os Princípios do Belo Web Design, 4ª Edição

Learn CSS in One Day and Learn It Well

Learn PHP in One Day and Learn It Well

Remote Jobs Become an affiliate Advertise

## Connect