



# Ignite



ReactJS



## Desafio 01 - Criando um hook de carrinho de compras

### [Sobre o desafio](#)

[Template da aplicação](#)

[Se preparando para o desafio](#)

[Fake API com JSON Server](#)

[Preservando carrinho com localStorage API](#)

[Mostrando erros com toastify](#)

[O que devo editar na aplicação?](#)

[components/Header/index.tsx](#)

[pages/Home/index.tsx](#)

[pages/Cart/index.tsx](#)

[hooks/useCart.tsx](#)

[Especificação dos testes](#)

[Como deve ficar a aplicação ao final?](#)

### [21 Entrega](#)



## Sobre o desafio

Nesse desafio, você deverá criar uma aplicação para treinar o que aprendeu até agora no ReactJS

Essa será uma aplicação onde o seu principal objetivo é criar um hook de carrinho de compras. Você terá acesso a duas páginas, um componente e um hook para implementar as funcionalidades pedidas nesse desafio:

- Adicionar um novo produto ao carrinho;
- Remover um produto do carrinho;
- Alterar a quantidade de um produto no carrinho;
- Cálculo dos preços sub-total e total do carrinho;
- Validação de estoque;
- Exibição de mensagens de erro;
- Entre outros.

A seguir veremos com mais detalhes o que e como precisa ser feito 🚀

## Template da aplicação

Para te ajudar nesse desafio, criamos para você esse modelo que você deve utilizar como um template do GitHub.

O template está disponível na seguinte URL:

github.com

<https://github.com/rocketseat-education/ignite-template-reactjs-criando-um-hook-de-carrinho-de-compras>

**Dica:** Caso não saiba utilizar repositórios do GitHub como template, temos um guia em [nosso FAQ](#).

## Se preparando para o desafio

Para esse desafio, além dos conceitos vistos em aula utilizaremos algumas coisa novas para deixar a nossa aplicação ainda melhor. Por isso, antes de ir diretamente para o código do desafio, explicaremos um pouquinho de:

- Fake API com JSON Server;
- Preservar dados do carrinho com localStorage API;
- Mostrar erros com toastify.

### Fake API com JSON Server

Assim como utilizamos o MirageJS no módulo 2 para simular uma API com os dados das transações da aplicação dt.money, vamos utilizar o JSON Server para simular uma API que possui as informações dos produtos e do estoque.

Navegue até a pasta criada, abra no Visual Studio Code e execute os seguintes comandos no terminal:

```
yarn yarn server
```

Em seguida, você vai ver a mensagem:

```
) yarn server
yarn run v1.22.10
$ json-server server.json -p 3333

\{^_^}/ hi!

Loading server.json
Done

Resources
http://localhost:3333/stock
http://localhost:3333/products

Home
http://localhost:3333

Type s + enter at any time to create a snapshot of the database
```

Perceba que ele iniciou uma fake API com os recursos `/stock` e `/products` em `localhost` na porta `3333` a partir das informações do arquivo `server.json` localizado na raiz do seu projeto. Acessando essas rotas no seu navegador, você consegue ver o retorno das informações já em JSON:

```
1 // 20210304195717
2 // http://localhost:3333/stock
3
4 ▾ [
5   ▾ {
6     "id": 1,
7     "amount": 3
8   },
9   ▾ {
10    "id": 2,
11    "amount": 5
12  },
13  ▾ {
14    "id": 3,
15    "amount": 2
16  },
17  ▾ {
18    "id": 4,
19    "amount": 1
20  },
21  ▾ {
22    "id": 5,
23    "amount": 5
24  },
25  ▾ {
26    "id": 6,
27    "amount": 10
28  }
29 ]
```

```
1 // 20210304195718
2 // http://localhost:3333/products
3
4 ▾ [
5   ▾ {
6     "id": 1,
7     "title": "Tênis de Caminhada Leve Confortável",
8     "price": 179.9,
9     "image": "https://rocketseat-cdn.s3-sa-east-1.amazonaws.com/modulo-redux/tenis1.jpg"
10  },
11  ▾ {
12    "id": 2,
13    "title": "Tênis VR Caminhada Confortável Detalhes Couro Masculino",
14    "price": 139.9,
15    "image": "https://rocketseat-cdn.s3-sa-east-1.amazonaws.com/modulo-redux/tenis2.jpg"
16  },
17  ▾ {
18    "id": 3,
19    "title": "Tênis Adidas Duramo Lite 2.0",
20    "price": 219.9,
21    "image": "https://rocketseat-cdn.s3-sa-east-1.amazonaws.com/modulo-redux/tenis3.jpg"
22  },
23  ▾ {
24    "id": 5,
25    "title": "Tênis VR Caminhada Confortável Detalhes Couro Masculino",
26    "price": 139.9,
27    "image": "https://rocketseat-cdn.s3-sa-east-1.amazonaws.com/modulo-redux/tenis2.jpg"
28  },
29  ▾ {
30    "id": 6,
31    "title": "Tênis Adidas Duramo Lite 2.0",
32    "price": 219.9,
33    "image": "https://rocketseat-cdn.s3-sa-east-1.amazonaws.com/modulo-redux/tenis3.jpg"
34  },
35  ▾ {
36    "id": 4,
37    "title": "Tênis de Caminhada Leve Confortável"
```

Dessa forma, basta consumir essas rotas da API normalmente com o axios. Caso queira estudar mais sobre o **JSON Server**, dê uma olhada aqui:

typicode/json-server

Get a full fake REST API with zero coding in less than 30 seconds (seriously) Created with public/index.html json-server

 <https://github.com/typicode/json-server>



## Preservando carrinho com localStorage API

Para preservar os dados do carrinho mesmo se fecharmos a aplicação, utilizaremos a **localStorage API**

Essa é uma API que nos permite persistir dados no navegador em um esquema de chave-valor (semelhante ao que temos com objetos JSON). Como essa é uma API global, você não precisa importar nada antes de usar.

Para salvar os dados, você deve utilizar o método `setItem`. Como primeiro argumento você deve informar o nome que você quer dar para o registro, no caso desse desafio é **obrigatório** utilizar o nome `@RocketShoes:cart`. Já o segundo argumento é o valor do registro que **obrigatoriamente** precisa estar no formato `string`. Abaixo segue um exemplo:

```
localStorage.setItem('@RocketShoes:cart', cart)
```

💡 Caso queira enviar um valor para o registro que não esteja no formato `string`, é preciso tratá-lo (ex.: `JSON.stringify`). Isso fará com que um objeto, lista, número ou qualquer outro valor seja convertido para uma string.

Para recuperar os dados, você deve utilizar o método `getItem` passando como argumento do registro que, no caso desse desafio, é **obrigatório** utilizar como `@RocketShoes:cart`. Abaixo segue um exemplo:

```
const storedCart = localStorage.getItem('@RocketShoes:cart');
```

💡 O valor retornado pelo método `getItem` é sempre no formato `string`. Caso você queira utilizar esse dado em outro formato, é preciso tratá-los (ex.: `JSON.parse`). Isso irá converter a informação ao estado original de quando foi salva com o `JSON.stringify`, seja uma lista, um objeto ou outro tipo de dado.

Caso queira estudar mais sobre a **localStorage API**, dê uma olhada aqui

### Window.localStorage

A propriedade `localStorage` permite acessar um objeto `Storage` local. A `localStorage` é similar ao `sessionStorage`. A única diferença é que enquanto os dados armazenados no `localStorage` não expiram,

 <https://developer.mozilla.org/pt-BR/docs/Web/API/Window/localStorage>

# Mostrando erros com toastify

Para mostrar os erros em tela, iremos utilizar um lib chamada **react-toastify**. Ela ajuda a mostra informações temporárias e rápidas de uma forma bem bonita.

De todos os métodos, utilizaremos apenas o `error` e será obrigatório utilizar mensagens predefinidas para que os testes passem (veremos mais sobre isso)

Caso queira estudar mais sobre a **react-toastify**, dê uma olhada aqui

fkhadra/react-toastify

🎉 React-Toastify allows you to add notifications to your app with ease. No more nonsense! \$ npm install --save react-toastify

🔗 <https://github.com/fkhadra/react-toastify#readme>



## O que devo editar na aplicação?

Com o template já clonado, as dependências instaladas e a fake API rodando, você deve completar onde não possui código com o código para atingir os objetivos de cada teste. Os documentos que devem ser editados são:

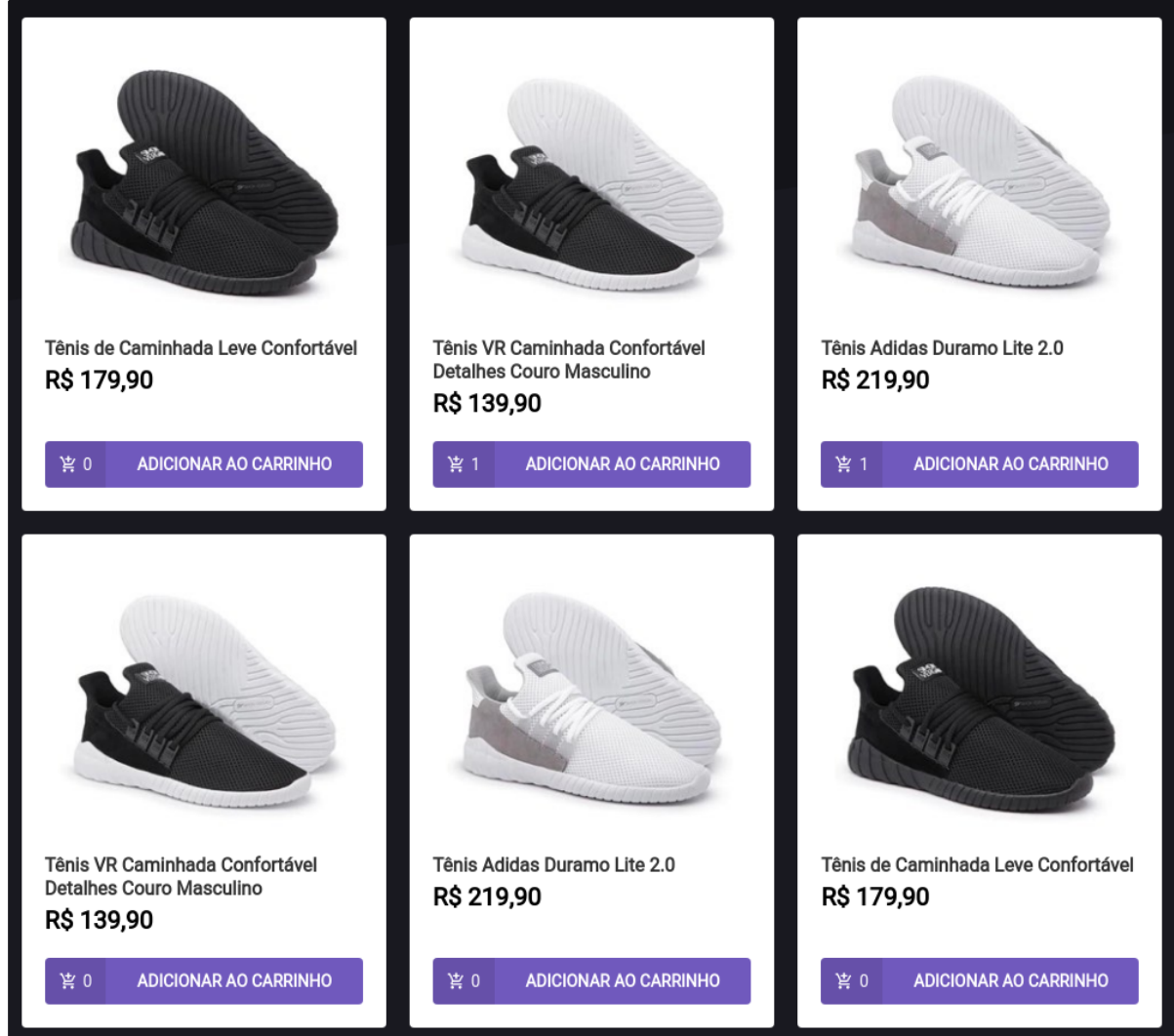
- [src/components/Header/index.tsx](#);
- [src/pages/Home/index.tsx](#)
- [src/pages/Cart/index.tsx](#);
- [src/hooks/useCart.tsx](#).

### components/Header/index.tsx



Você deve receber o array `cart` do hook `useCart` e mostrar em tela a quantidade de produtos **distintos** adicionados ao carrinho. Dessa forma, se o carrinho possui 4 unidades do item A e 1 unidade do item B o valor a ser mostrado é `2 itens`.

### pages/Home/index.tsx





Você deve renderizar os produtos buscados da fake API em tela com as informações de título, imagem, preço, quantidade adicionada ao carrinho. Por fim, é preciso implementar a funcionalidade de adicionar o produto escolhido ao carrinho ao clicar no botão **ADICIONAR AO CARRINHO**.

Nesse arquivo, temos três pontos importantes a serem implementados:

- **cartItemsAmount:** Deve possuir as informações da quantidade de cada produto no carrinho. Sugerimos criar um array chave/valor utilizando **reduce** onde a chave representa o id do produto e o valor a quantidade do produto no carrinho.
- **loadProducts:** Deve buscar os produtos da Fake API e formatar o preço utilizando o helper **utils/format**
- **handleAddProduct:** Deve adicionar o produto escolhido ao carrinho.

**pages/Cart/index.tsx**

PRODUTO	QTD	SUBTOTAL
 <div> Tênis VR Caminhada Confortável Detalhes Couro Masculino  R\$ 139,90 </div>	<div> <div>⊖</div> <div>2</div> <div>⊕</div> </div>	R\$ 279,80 <div>🗑</div>
 <div> Tênis Adidas Duramo Lite 2.0  R\$ 219,90 </div>	<div> <div>⊖</div> <div>1</div> <div>⊕</div> </div>	R\$ 219,90 <div>🗑</div>
<div>FINALIZAR PEDIDO</div>		TOTAL <b>R\$ 499,70</b>

Você deve renderizar uma tabela com a imagem, título, preço unitário, quantidade de unidades e preço subtotal de cada produto o carrinho. Além disso, também é preciso renderizar o preço total do carrinho. Por fim, é preciso implementar as funcionalidades dos botões de decrementar, incrementar e remover o produto do carrinho.

Nesse arquivo, temos cinco pontos importantes a serem implementados:

- **cartFormatted:** Deve formatar o carrinho adicionando os campos `priceFormatted` (preço do produto) e `subTotal` (preço do produto multiplicado pela quantidade) ambos devidamente formatados com o `utils/format`.
- **total:** Deve possuir a informação do valor total do carrinho devidamente formatado com o `utils/format`.
- **handleProductIncrement:** Deve aumentar em 1 unidade a quantidade do produto escolhido ao carrinho.
- **handleProductDecrement:** Deve diminuir em 1 unidade a quantidade do produto escolhido ao carrinho, onde o valor mínimo é 1 (nesse caso o botão deve estar desativado).
- **handleRemoveProduct:** Deve remover o produto escolhido do carrinho.

## hooks/useCart.tsx

Apesar de não retornar diretamente nenhuma renderização de elementos na interface como os outros arquivos, esse é o coração do desafio. Ele é responsável por:

- hook `useCart`;
- context `CartProvider`;
- manipular `localStorage`;
- exibir `toasts`.

Então é aqui que você vai implementar as funcionalidades que serão utilizadas pelo restante do app. Os principais pontos são:

- **cart:** Deve verificar se existe algum registro com o valor `@RocketShoes:cart` e retornar esse valor caso existir. Caso contrário, retornar um array vazio.
- **addProduct:** Deve adicionar um produto ao carrinho. Porém, é preciso verificar algumas coisas:
  - O valor atualizado do carrinho deve ser perpetuado no **localStorage** utilizando o método `setItem`.
  - Caso o produto já exista no carrinho, não se deve adicionar um novo produto repetido, apenas incrementar em 1 unidade a quantidade;
  - Verificar se existe no estoque a quantidade desejada do produto. Caso contrário, utilizar o método `error` da **react-toastify** com a seguinte mensagem:

```
toast.error('Quantidade solicitada fora de estoque');
```

- Capturar utilizando `trycatch` os erros que ocorrerem ao longo do método e, no catch, utilizar o método `error` da **react-toastify** com a seguinte mensagem:

```
toast.error('Erro na adição do produto');
```

- **removeProduct:** Deve adicionar um produto ao carrinho. Porém, é preciso verificar algumas coisas:
  - O valor atualizado do carrinho deve ser perpetuado no **localStorage** utilizando o método `setItem`.
  - Capturar utilizando `trycatch` os erros que ocorrerem ao longo do método e, no catch, utilizar o método `error` da **react-toastify** com a seguinte mensagem:

```
toast.error('Erro na remoção do produto');
```



- **updateProductAmount:** Deve adicionar um produto ao carrinho. Porém, é preciso verificar algumas coisas:
  - O valor atualizado do carrinho deve ser perpetuado no **localStorage** utilizando o método `setItem`.
  - Se a quantidade do produto for menor ou igual a zero, sair da função **updateProductAmount** instantaneamente.
  - Verificar se existe no estoque a quantidade desejada do produto. Caso contrário, utilizar o método `error` da **react-toastify** com a seguinte mensagem:

```
toast.error('Quantidade solicitada fora de estoque');
```

- Capturar utilizando `trycatch` os erros que ocorrerem ao longo do método e, no catch, utilizar o método `error` da **react-toastify** com a seguinte mensagem:

```
toast.error('Erro na alteração de quantidade do produto');
```

## Especificação dos testes

Em cada teste, tem uma breve descrição no que sua aplicação deve cumprir para que o teste passe.



Caso você tenha dúvidas quanto ao que são os testes, e como interpretá-los, dê uma olhada em [nosso FAQ](#)

Para esse desafio, temos os seguintes testes:



[Teste components/Header/index.tsx](#)



[Testes pages/Home/index.tsx](#)



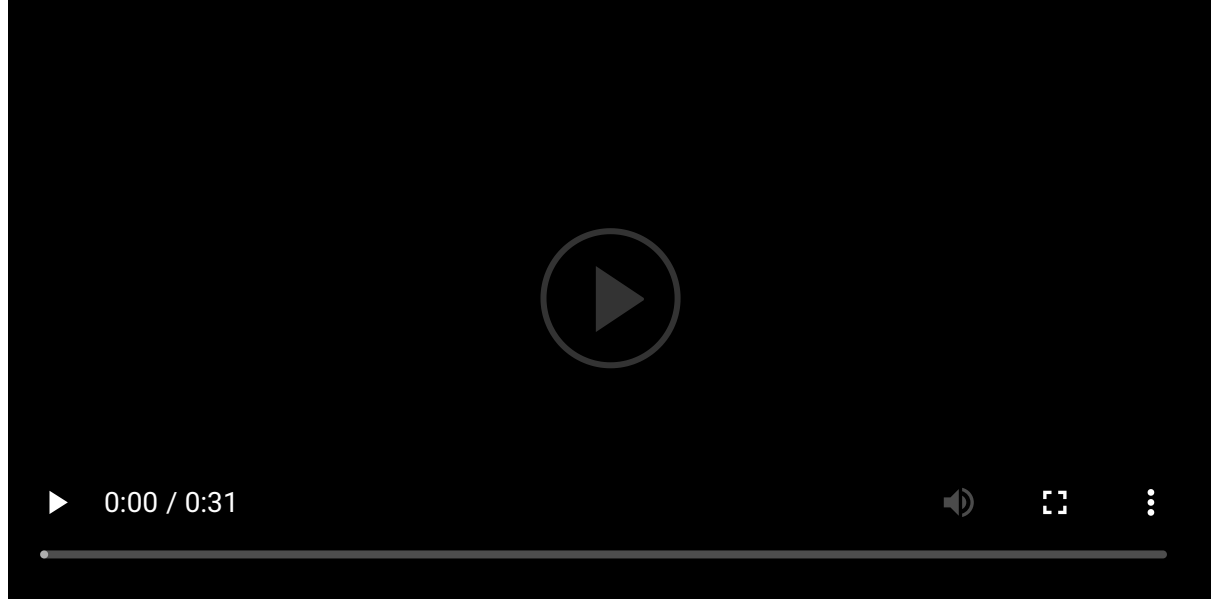
[Testes pages/Cart/index.tsx](#)



[Testes hooks/useCart.tsx](#)

## Como deve ficar a aplicação ao final?

Está com dúvidas (ou curioso 🧐) para ver como deve ficar a aplicação ao final do desafio? Deixamos abaixo um vídeo mostrando as principais funcionalidades que você deve implementar para te ajudar (ou matar sua curiosidade 🧐).



## Entrega

Esse desafio deve ser entregue a partir da plataforma da Rocketseat. Envie o link do repositório que você fez suas alterações. Após concluir o desafio, além de ter mandado o código para o GitHub, fazer um post no LinkedIn é uma boa forma de demonstrar seus conhecimentos e esforços para evoluir na sua carreira para oportunidades futuras.

Feito com 💜 por Rocketseat 🙌 Participe da nossa [comunidade aberta!](#)