FRAMEWORKS FULL STACK

Texto base

5

Criação de carrinho de compras

Prof. André Nascimento Maia
Prof. Caio Nascimento Maia

Resumo

Entender a estrutura de um projeto que utiliza o framework Next.js e quais as formas recomendadas para criação de componentes React são fundamentais para garantir a produtividade e eficiência no desenvolvimento de aplicações front-end. Utilizaremos um contexto de criação de um carrinho de compras para aplicação de front-end de um e-commerce para apresentar esses conceitos.

5.1. Introdução

O desenvolvimento de *software* é sempre muito mais claro quando estamos trabalhando em um contexto rico e completo para demonstração de diversas técnicas e análises. Neste texto, o contexto é a construção de um carrinho de compras para um e-commerce. A partir desse contexto, analisaremos qual seria uma estrutura adequada para um projeto de *front-end* utilizando o *framework* Next.js e também discutiremos os conceitos recomendados pela comunidade React para a construção de componentes.

Não existem arquiteturas e modelos ideais em engenharia de software. Sempre que tivermos benefícios em alguma decisão que foi tomada ao projetar o sistema, também teremos inconvenientes ligados à decisão tomada. O nosso objetivo é sempre maximizar os benefícios e minimizar os inconvenientes. Desenvolver utilizando uma abordagem *full-stack* é sempre necessário considerarmos o benefício de maior produtividade e assertividade no desenvolvimento.

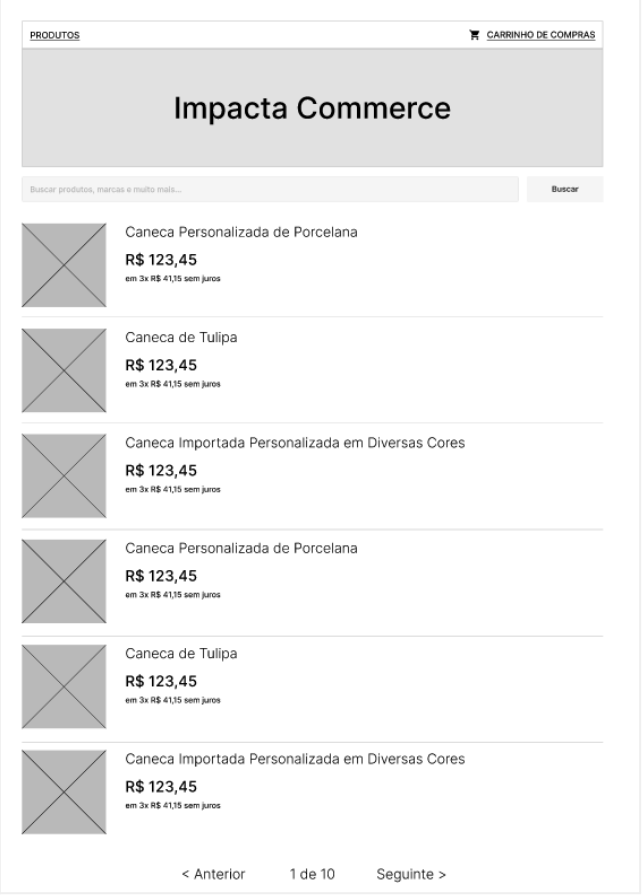
5.2 Estrutura para projeto Next.js

Não existe uma estrutura ideal pré-estabelecida pelo *framework* Next.js para qualquer tipo de aplicação de *front-end*. Porém, existem algumas convenções sobre nomes de arquivos e diretórios e sugestões de como organizar os arquivos da aplicação para que tenhamos facilidade durante manutenções e adição de novas funcionalidades.

Como exemplo, considere que tenhamos que desenvolver um e-commerce com pelo menos 4 páginas diferentes:

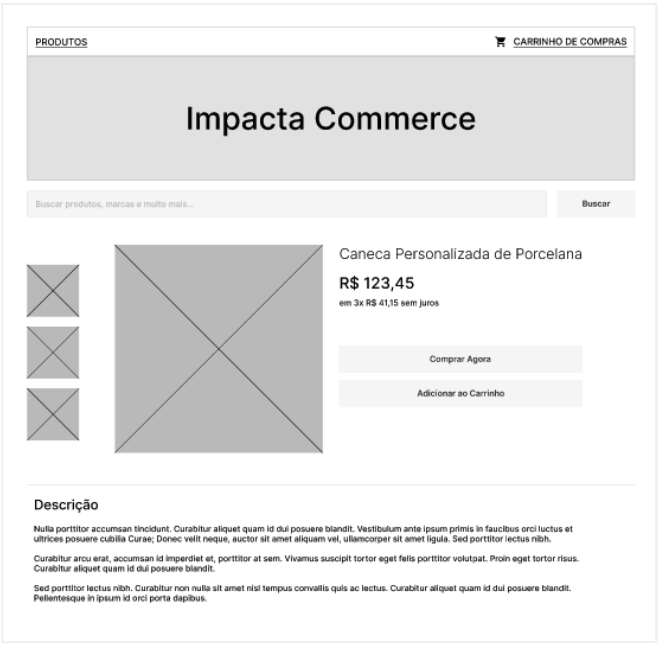
1. **Busca e listagem de projetos (Figura 5.1)**. Uma lista de produtos paginada, com cada item da lista contendo foto do produto, título, valor e valor parcelado com ou sem juros, além de um campo de busca em texto livre para filtrar os produtos disponíveis para venda;
2. **Detalhes de um produto (Figura 5.2)**. Apresenta os detalhes de um produto, que contém imagens, título, valor, valor parcelado com ou sem juros, descrição detalhada em texto e botões de ação para comprar agora ou adicionar ao carrinho.
3. **Carrinho de compras (Figura 5.3)**. Uma lista com todos os produtos que foram adicionados ao carrinho do cliente, contendo quantidades e totalizadores.
4. **Finalização do pedido (Figura 5.4)**. Mensagem padrão de pedido finalizado informando ao usuário que essa é uma aplicação de exemplo e que nenhum pedido será entregue.

Figura 5.1: protótipo de tela de busca e listagem de produtos



Fonte: autores, 2022

Figura 5.2: protótipo de tela de detalhes de um produto.

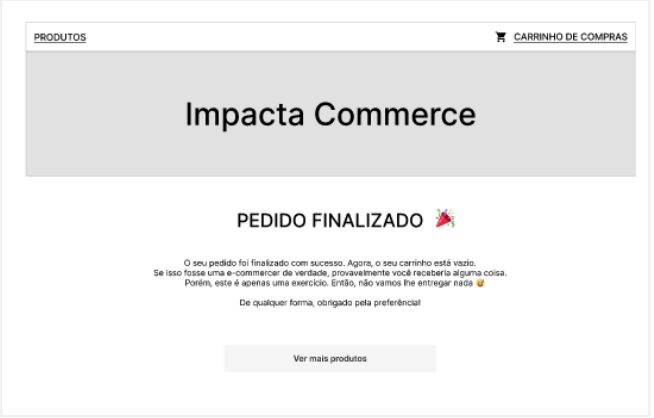


Fonte: autores, 2022

Figura 5.3: protótipo de tela de carrinho de compras

Fonte: autores, 2022

Figura 5.4: protótipo de tela de finalização do pedido



Fonte: autores, 2022

Utilizando Next.js para criação dessa aplicação de *front-end*, em todas as páginas teremos diversos componentes React, sendo alguns componentes específicos da aplicação com o objetivo de exibição e controle de informações específicas, como detalhes de um produto e galeria de fotos do produto, mas também considere que outros componentes são estruturais, como o *layout* de cada página, rodapés de páginas e botões. Fora as páginas e componentes, também teremos arquivos estáticos como imagens, folhas de estilos, e arquivos de configurações para o *framework*.

Como sugestão, vamos utilizar a organização de arquivos e diretórios conforme sugerido no Código 5.1.

A estrutura proposta no Código 5.1 segue os mesmos padrões da aplicação de exemplo disponível na documentação de exemplo do Next.js (Vercel, 2022). Existem diversos outros modelos e padrões a serem seguidos (Stack Overflow, 2018), e cada um dos padrões e modelos propostos têm benefícios e inconvenientes envolvidos que afetarão o futuro do projeto. É importante ressaltar que não é necessário identificar o melhor padrão ou modelo de imediato. Procurar o melhor modelo exige tempo e muita análise, além disso, o escopo e necessidades do projeto podem mudar com o passar do tempo e tornar todos os benefícios da escolha em inconvenientes. Então, a regra de ouro para situações como estas é escolher o melhor para o momento e com o futuro que conseguimos enxergar para a aplicação que estamos desenvolvendo. Se for necessário alterar a estrutura no futuro, apenas altere e siga em frente.

Código 5.1: Exemplo de estrutura de diretórios e arquivos para um projeto Next.js.

```
.
├── README.md
├── package-lock.json
├── package.json
├── lib
│   ├── constants.js
│   └── api.js
├── components
│   └── ProductsForSaleList.js
├── pages
│   ├── cart.js
│   ├── orders
│   │   └── [code].js
│   └── products
│       ├── [code].js
│       └── index.js
├── public
│   ├── favicon.ico
│   └── vercel.svg
└── styles
    └── index.css
```

Fonte: autores, 2022.

Essa estrutura, embora seja simples, é muito eficiente e clara. Temos um arquivo para instruções sobre o projeto chamado README.md, os arquivos de configuração dos módulos do Node.js package*.json. O diretório lib, tem como objetivo armazenar todos os scripts necessários para comunicação via API REST, constantes da aplicação ou qualquer outro script necessário para lidar com regras de negócio e manipulação de estruturas de dados. Quando criarmos componentes React, armazenaremos todos eles no diretório components; tanto os componentes estruturais quanto os específicos para a aplicação. No nosso exemplo, já temos um componente chamado ProductsForSaleList que é responsável pela lista de produtos representada no protótipo da Figura 5.1. Todas as páginas da aplicação serão armazenadas no diretório chamado *pages*, sendo um script Javascript para cada página seguindo o padrão de URL que queremos que o usuário visualize em seu navegador. O diretório *public* contém todos os arquivos estáticos como imagens, ícones, fontes, e outros. Também temos o diretório *styles* que contém todas as folhas de estilo globais aplicadas à aplicação.

5.2.1. O diretório pages em Next.js

Uma das principais características do *framework* Next.js é a capacidade de roteamento de páginas baseado no sistema de arquivos. Isso permite maior facilidade e flexibilidade na criação de novas páginas. Uma página em Next.js é um componente React armazenado no diretório *pages*.

Cada uma das páginas é associada com uma rota gerada automaticamente. No Código 5.1 utilizamos três tipos de roteamento: rota de índice, rota aninhado e rota de segmentos dinâmicos.

A rota de índice, é padrão e mais utilizado. Toda vez que um arquivo tiver o nomes de index.js, por exemplo, a rota criada para ele será o mesmo que o caminho para o diretório que o arquivo index.js está contido. Alguns exemplos de roteamento baseado em índices estão presentes na Tabela 5.1.

Tabela 5.1. Exemplos de rotas de índice.

Caminho de diretórios	Resultado de roteamento
pages/index.js	/
pages/products/index.js	/products
pages/orders/index.js	/orders

Fonte: autores, 2022.

Às vezes, é necessário criar uma rota complexa com vários níveis, estas são chamadas de rotas aninhadas. A Tabela 5.2 mostra alguns exemplos de rotas aninhadas.

Tabela 5.2. Exemplos de rotas aninhadas

Caminho de diretórios	Resultado de roteamento
pages/blog/post-principal.js	/blog/post-principal
pages/settings/profile.js	/settings/profile
pages/dir1/dir2/dir3/admin.js	/dir1/dir2/dir3/admin

Fonte: autores, 2022.

Frequentemente, informações específicas são passadas em um dos segmentos da URL como código de produtos e usuários. Utilizamos esse tipo de roteamento para a estrutura de arquivos sugerida para o e-commerce para obter os detalhes de um produto disponível para venda e também para obter os detalhes de um pedido específico. A Tabela 5.3 mostra os exemplos que utilizamos e também outros exemplos fictícios.

Tabela 5.3. Exemplos de rotas de segmentos dinâmicos

Caminho de diretórios	Resultados de roteamento
pages/orders/[code].js	/orders/123 /orders/qualquer-texto /orders/ABC123
pages/products/[code].js	/products/ABC123
pages/[username]/[repositoryname]/pulls.js	/meu-user/repositoryAppABC/pulls

Fonte: autores, 2022.

A tabela 5.3 contém dois exemplos que utilizamos para o e-commerce e um fictício baseado na URL que o Github utiliza para exibir as Pull Requests de um repositório qualquer. Sempre que utilizamos colchetes no nome do arquivo ou no nome do diretório, o Next.js entende que nesse trecho do caminho do sistema de arquivos, o conteúdo deve ser dinâmico. Para acessar o segmento dinâmico dentro da página, devemos utilizar o *hook* chamado *useRoute* para acessar os segmentos na URL.

5.3. Pensando do jeito React para construção de componentes

A forma recomendada pelos criadores para a criação de componentes e aplicações utilizando o *framework* React é pensar de um jeito estruturado e seguir passos bem definidos para que se tenho o mínimo de retrabalho, manutenção e garantir uma boa organização para o projeto e o desenvolvimento de código (Meta Platforms, 2022). A ideia principal é separar o processo em dois momentos. O primeiro mantém o foco na exibição de uma versão estática, sem interatividade, de um modelo de dados, lidando com *props*, o que requer muito código e pouco pensamento. No segundo momento, mantém-se o foco exclusivamente na interatividade, lidando com *states*, o que requer muito pensamento e pouco código.

Os passos sugeridos para desenvolvimento de componentes (Meta Platforms, 2022) são:

1. Ter um protótipo da tela e do modelo de dados;
2. Decompor a interface de usuário em uma hierarquia de componentes;
3. Criar a versão estática em React;
4. Identificar a representação mínima do *state;*
5. Identificar onde o *state* deve ficar;
6. Adicionar o fluxo de dados inverso;

Vamos seguir os 6 passos para a construção do nosso carrinho de compras.

5.3.1 Passo 1 - Ter um protótipo da tela e do modelo de dados

Como protótipo, vamos utilizar o carrinho de compras da Figura 5.3. Então, nos falta apenas um modelo de dados suficiente para o carrinho de compras.

O Código 5.2 mostra um objeto chamado *cart* que contém uma lista de produtos com seus detalhes mínimos para exibição de acordo com o protótipo definido.

Código 5.2: Objeto *cart* em JSON utilizado como modelo de dados para o carrinho de compras de um cliente.

```
{
  "createdAt": "2022-01-01T14:38:33Z", // datas em ISO-8601
  "updatedAt": "2022-01-01T14:38:33Z", // datas em ISO-8601
  "products": [
    {
      "title": "Caneca Personalizada de Porcelana",
      "tumbnailUrl": "http://meu-ecommerce.com/img/pr/tb-pers-porc.jpg",
      "qty": 1,
      "unitPrice": 123.45
    },
    {
      "title": "Caneca Importada Personalizada em Diversas Cores",
      "tumbnailUrl": "http://meu-ecommerce.com/img/pr/tb-import-colors.jpg",
      "qty": 2,
      "unitPrice": 123.45
    },
    {
      "title": "Caneca de Tulipa",
      "tumbnailUrl": "http://meu-ecommerce.com/img/pr/tb-tuli.jpg",
      "qty": 1,
      "unitPrice": 123.45
    }
  ]
}
```

Fonte: autores, 2022.

O modelo de dados apresentado no Código 5.2 contém 3 informações principais. A primeira informação é a data de criação do carrinho e a segunda é a data de atualização. Ambas as informações estão formatadas em ISO-8601 para facilitar as diversas conversões possíveis para uma possível exibição. Para o protótipo atual do carrinho de compras, essas informações não são relevantes. A terceira informação, é a lista de produtos selecionados para o carrinho. Estes são de extrema relevância e cada um dos objetos produtos dentro da lista *products* contém exatamente as informações que precisamos exibir de acordo com o protótipo da Figura 5.3.

5.3.2 Passo 2 - Decompor a interface de usuário em uma hierarquia de componentes

A partir do protótipo de tela de carrinho de compras da Figura 5.3, vamos decompor a interface de usuário em uma hierarquia de componentes adequada a nossa

aplicação. Lembre-se que cada componente é uma função Javascript, então é recomendado (Meta Platforms, 2022) que você utilize o princípio da responsabilidade única (Martin et al., 2003, 95) para decompor os componentes e subcomponentes.

A partir do protótipo do carrinho de compras da Figura 5.3, foram marcados e numerados os componentes decompostos. A Figura 5.4 servirá de base para elencar os componentes e subcomponentes que devem ser criados.

Figura 5.5: Protótipo de carrinho de compras decomposto em componentes React.

Fonte: autores, 2022.

Em detalhes, os números da imagem correspondem a lista de componentes conforme abaixo.

1. **Cart (vermelho)**: componente completo para o carrinho de compras;
2. **ItemsList (amarelo)**: uma lista de itens do carrinho de compras;
3. **Item (azul)**: um item do carrinho de compras, no caso, um produto com foto, título, quantidade e valor total;
4. **Summary (verde)**: o valor total calculado para o carrinho de compras;
5. **ItemQuantity (roxo)**: componente dinâmico para exibir e atualizar a quantidade de produtos.

Após decompostos os componentes, o próximo passo é organizá-los em uma hierarquia de componente. Nesse caso, a hierarquia seria:

- Cart
    - ItemsList
        - Item
            - ItemQuantity
    - Summary

5.3.3 Passo 3 - Criar a versão estática em React

Com os componentes e a hierarquia definidos, o próximo passo é criar uma versão estática dos componentes, ou seja, sem interatividade, utilizando apenas *props* ao invés de *state* e passando os valores sobre um modelo fixo.

Os Código 5.3, 5.4, 5.5, 5.6 e 5.7 mostram o código necessário para a criação de cada um dos componentes de acordo com a hierarquia estabelecida em uma versão sem interatividade, estática.

Código 5.3: Componente Cart.

```
01. import ItemsList from "./ItemsList";
02. import Summary from "./Summary";
03.
04. function Cart(props) {
05.   return (
06.     <div>
07.       <hr/>
08.       <ItemsList products={props.cart.products} />
09.       <hr/>
10.       <Summary products={props.cart.products} />
11.     </div>
12.   );
13. }
14.
15. export default Cart;
```

Fonte: autores, 2022.

Código 5.4: Componente ItemList

```
01. import Item from "./Item";
02.
03. function ItemsList(props) {
04.
05.   let rows = [];
06.
07.   props.products.forEach((product, index) => {
08.     rows.push(<Item key={index} product={product} />);
09.   });
10.
11.   return <div>{rows}</div>;
12. }
13.
14. export default ItemsList;
```

Fonte: autores, 2022.

Código 5.5: Componente Summary.

```
01. function Summary(props) {
02.   const subtotal = props.products
03.     .map((x) => x.unitPrice * x.qty)
04.     .reduce((prev, e) => prev + e, 0);
05.
06.   return (
07.     <div className="container">
08.       <div className="row">
09.         <div className="col-6"></div>
10.         <div className="col-4 text-end">SUBTOTAL</div>
11.         <div className="col-2 text-end">
12.           <h5>R$ {subtotal}</h5>
13.         </div>
14.       </div>
15.     </div>
16.   );
17. }
18.
19. export default Summary;
```

Fonte: autores, 2022.

Código 5.6: Componente Item.

```
01. import ItemQuantity from "./ItemQuantity";
02.
03. function Item(props) {
04.   const defaultProductImage = "https://via.placeholder.com/150";
05.
06.   return (
07.     <div className="container my-2">
08.       <div className="row">
09.        <div className="col-2">
10.        <img src={defaultProductImage} className="img-thumbnail" />
11.        </div>
12.         <div className="col-6 py-5">
13.           <h5>{props.product.title}</h5>
14.         </div>
15.         <div className="col-2 py-5">
16.           <ItemQuantity product={props.product} />
17.         </div>
18.         <div className="col-2 py-5 text-end">
19.           <h4>R$ {props.product.unitPrice * props.product.qty}</h4>
20.         </div>
21.       </div>
22.     </div>
23.   );
24. }
25.
26. export default Item;
```

Fonte: autores, 2022.

Código 5.7: Componente.

```
01. function ItemQuantity(props) {
02.   return (
03.     <div className="input-group mb-3">
04.       <button className="btn btn-outline-secondary" type="button">
05.         -
06.       </button>
07.       <input
08.         type="number"
09.         className="form-control text-center"
10.       />
11.       <button className="btn btn-outline-secondary" type="button">
12.         +
13.       </button>
14.     </div>
15.   );
16. }
17. export default ItemQuantity;
```

Fonte: autores, 2022.

Após criar todos esses componentes, adicionar o componente Cart a página pages/cart informando a propriedade *cart* que deve ser um objeto no modelo estabelecido em Código 5.2, o resultado é uma página cujo o conteúdo do carrinho de compra é igual ao proposto no protótipo da Figura 5.3.

5.3.4 Passo 4 - Identificar a representação mínima do state

Para identificar o que deve ser um *state*, verificamos todos os dados que são exibidos e toda a interatividade que temos no componente. Para o carrinho de compra proposto, temos:

- Lista de itens do carrinho;
- Quantidade de cada item do carrinho;
- Valor total de cada item;
- Valor total do carrinho.

Os especialistas em React sugerem 3 perguntas para identificar se uma informação faz parte do *state* (Meta Platforms, 2022), são elas:

1. O valor é recebido pelo por *props*? Se sim, provavelmente não é *state*.
2. O valor se mantém inalterado ao longo do tempo? Se sim, provavelmente não é *state*.
3. O valor pode ser computado através de qualquer outro *state* ou props do seu componente? Se sim, não é *state*.

A lista de itens do carrinho é passada via props e se mantém inalterada, então não faz parte do *state*. A quantidade de cada item do carrinho tem um valor inicial, mas permite interatividade, ou seja, não se mantém inalterado, então faz parte do *state*. O valor total de cada item e o valor total do carrinho não fazem parte do *state* porque eles podem ser computados a partir do *props* e *state* passado pelo componente pai.

Dessa forma, nosso *state* é composto apenas da quantidade de cada item do carrinho.

5.3.5 Passo 5 - Identificar onde o state deve ficar

O *state* deve ficar no componente pai imediatamente acima dos componentes que são atualizados com a interatividade.

Como o objeto alterado é a lista de produtos, é necessário elevar o *state* do componente mais interno para o componente que detém o *state* (Meta Platforms, 2022). Suponha que em nosso caso o componente que detém o *state* é o componente Cart, o mais alto nível. Para elevar o *state* de ItemQuantity até Cart, será necessário criar uma nova *prop* que chamaremos de *onQuantityChanged* e que tem como argumento um objeto com 2 atributos. O primeiro é o produto que está sendo alterado, o segundo é a nova quantidade para o determinado produto. O código atual de ItemQuantity (Código 5.7) deve ser alterado para Código 5.8.

Código 5.8: Elevando o *State* de ItemQuantity.

```
01. function ItemQuantity(props) {
02.   function buildQuantityEventChanged(increment) {
03.     return {
04.       product: props.product,
05.       newQty: props.product.qty + increment,
06.     };
07.   }
08.
09.   function decrement() {
10.     props.onQuantityChanged(buildQuantityEventChanged(-1));
11.   }
12.
13.   function increment() {
14.     props.onQuantityChanged(buildQuantityEventChanged(+1));
15.   }
16.
17.   function handleChange(e) {
18.     props.onQuantityChanged(
19.       buildQuantityEventChanged(e.target.value - props.product.qty)
20.     );
21.   }
22.
23.   return (
24.     <div className="input-group mb-3">
25.       <button className="btn btn-outline-secondary" type="button"
26.         onClick={decrement}>-</button>
27.       <input type="number" className="form-control text-center"
28.         value={props.product.qty} onChange={handleChange} />
29.       <button className="btn btn-outline-secondary" type="button"
30.         onClick={increment} >+</button>
31.     </div>
32.   );
33. }
34. export default ItemQuantity;
```

Fonte: autores, 2022.

Alguns eventos para controlar os cliques dos botões de incremento e decremento foram adicionados (linhas 26 e 30 do Código 5.8). Todas as funções para controlar a manipulação de controles, delegam a chamada para a função *props.onQuantityChanged* passada do componente mais alto nível (Cart) até o componente ItemQuantity.

5.3.6 Passo 6 - Adicionar o fluxo de dados inverso

Quando preenchemos o *state* com a lista de itens do Cart, a informação é repassada para os componentes internos da hierarquia como *props*. Porém, se você tentar alterar o conteúdo do *input* de quantidade diretamente sem implementar o evento *onChange*, perceberá que não será possível alterar a sua quantidade. Isso acontece porque o React garante que se um *input value* está mapeado para uma *props* gerenciada pelo seu contexto, o *input value* só será alterado quando a *props* ou *state* do contexto do React for alterada.

No Código 5.8, linha 28, foi adicionado o evento *handleChange* para capturar alterações manuais diretamente através do evento *onChange* no *input* que armazena a quantidade de cada item. A única coisa que o *handleChange* faz, é o mesmo que as funções *increment* e *decrement* fazem, envia um novo evento elevando o *state* para que o *state* gerenciado pelo Cart seja atualizado.

5.4. Conclusões

Não há uma estrutura ou modelo de organização padrão para um projeto que utiliza Next.js. Alguns diretórios são convencionados para facilitar e não precisar nenhum tipo de configuração adicional, como é o caso do diretório *pages*. Independentemente de se ter selecionado a melhor organização para um projeto, isso não deve ser algo que bloqueie o processo de desenvolvimento, tornando-o ineficiente. Siga sempre com a sugestão que melhor fizer sentido para o momento e o contexto entendido para a aplicação, se necessário, atualize a organização no futuro.

Embora a criação de componentes em React seja simples e enxuta, um bom planejamento e visão da hierarquia dos componentes, seguido de um bom entendimento sobre o momento de exibição e o de interatividade do usuário, podem contribuir significativamente na organização e produtividade na criação de componentes. Caso contrário é bem provável que em uma estrutura complexa de componentes, muitos problemas comecem a surgir e diminuam a produtividade.

Como sempre, um bom planejamento pragmático é a chave para desenvolvimento eficiente e produtivo utilizando Next.js e React.

Referências

MARTIN, R. C; CHANDRAKASAN, A. P; NIKOLI*C, B; RABAEY, J. M. (2003). **Agile software development**: principles, patterns, and practices. Pearson Education.

META PLATFORMS. (2022). **Elevando o state** – react. react. Disponível em: <https://pt-br.reactjs.org/docs/lifting-state-up.html>. Acessado em: 01 fev. 2022.

META PLATFORMS. (2022). **Pensando do jeito react** – react. React. Disponível em: <https://pt-br.reactjs.org/docs/thinking-in-react.html>. Acessado em: 03 fev. 2022.

STACK OVERFLOW. (2018). **Is this Next.JS folder structure recommended?** Stack Overflow. Disponível em: <https://stackoverflow.com/questions/53854104/is-this-next-js-folder-structure-recommended>. Acessado em: 03 fev. 2022.

VERCEL. (2022). **cms-wordpress**. GitHub. Disponível em: <https://github.com/vercel/next.js/tree/canary/examples/cms-wordpress>. Acessado em: 03 fev. 2022.

VERCEL. (2022). **Routing:** dynamic routes. Next.js. Disponível em: <https://nextjs.org/docs/routing/dynamic-routes>. Acessado em: 03 fev. 2022.