

INTENSIVÃO DE JAVASCRIPT

Apostila Completa Aula 5

Guia passo a passo para construir seu próprio site
de E-Commerce a partir do zero!



Parte 1

Carrinho de compras – Totalizando preços



Carrinho de compras – Totalizando Preços

O nosso carrinho de compras está quase pronto, mas ainda precisamos implementar algumas funcionalidades e dinamismo na área de Preço total dos produtos e o botão de finalizar.

A primeiro momento vamos focar no Preço total. Dentro do arquivo index.html, o nosso **elemento <p>** de **id "preco-total"** é o responsável por armazenar as informações sobre o total de preço. Então observe que colocamos um valor fixo nesse elemento e vamos alterar para o texto – "Total:", a parte '\$200' após aplicarmos a lógica Javascript será alterado e teremos o nosso valor retornado dinamicamente.

```
<p id="preco-total">Total: $200</p>
```

E agora vamos trabalhar no **arquivo menuCarrinho.js**, que é o local onde fica a lógica do nosso Preço total.

Vamos criar uma função chamada **atualizarPecoCarrinho()** e dentro dela iremos aplicar a lógica que queremos para que o elemento atualize os preços dinamicamente.

Carrinho de compras – Totalizando Preços

```
function atualizarPrecoCarrinho() {  
  const precoCarrinho = document.getElementById("preco-total");  
  let precoTotalCarrinho = 0;  
  for (const idProdutoNoCarrinho in idsProdutoCarrinhoComQuantidade) {  
    precoTotalCarrinho += catalogo.find(p => p.id === idProdutoNoCarrinho).preco * idsProdutoCarrinhoComQuantidade[idProdutoNoCarrinho];  
  }  
  precoCarrinho.innerText = `Total: ${precoTotalCarrinho}`;  
}
```

O código acima é a implementação da lógica de Javascript dentro da nossa função **atualizarPrecoCarrinho()**. Vamos separar e explicar todo o conceito que foi utilizado para sua criação.

1- const precoCarrinho = document.getElementById("preco-total"); - Isso obtém o elemento HTML com o ID "preco-total", é o local onde desejamos mostrar o preço total do carrinho.

2- let precoTotalCarrinho = 0; - Isso inicializa uma variável chamada **precoTotalCarrinho** com o valor inicial de 0. Esta variável será usada para acumular o preço total dos produtos no carrinho.

3- for (const idProdutoNoCarrinho in idsProdutoCarrinhoComQuantidade) { - Isso começa um loop **for...in** que percorre cada identificador de produto no objeto **idsProdutoCarrinhoComQuantidade**. Objeto que armazena os ids dos produtos e as quantidades de cada um deles.

Carrinho de compras – Totalizando Preços

4- `precoTotalCarrinho += catalogo.find(p => p.id === idProdutoNoCarrinho) * idsProdutoCarrinhoComQuantidade[idProdutoNoCarrinho];` – Neste trecho, estamos procurando no array **catalogo** um objeto de produto com um ID correspondente ao **idProdutoNoCarrinho**. Então, multiplicamos o preço desse produto (acessado por **.preco**) pela quantidade dele no carrinho (obtida através do **idsProdutoCarrinhoComQuantidade**). O resultado dessa multiplicação é adicionado ao **precoTotalCarrinho**. Basicamente, estamos calculando o preço total acumulado para todos os produtos no carrinho.

5- `precoCarrinho.innerHTML = Total: ${precoTotalCarrinho};` – Isso será responsável por atualizar o conteúdo do elemento `<p>` com o valor calculado **precoTotalCarrinho**, exibindo o preço total formatado com a mensagem "Total: \$" na frente.

Portanto, a função calcula o preço total do carrinho somando os preços individuais dos produtos multiplicados pelas suas quantidades e atualiza visualmente o elemento no HTML com o valor total calculado.

Carrinho de compras – Totalizando Preços

E agora precisamos utilizar essa função, isso será feito dentro das funções que atualizam os produtos do nosso carrinho de compras, ou seja, vamos adicionar as nossas funções o código "**atualizarPrecoCarrinho()**" em cada uma delas, que são:

- **Função renderizarProdutosCarrinho():**
- **Função incrementarQuantidadeProduto():**
- **Função decrementarQuantidadeProduto():**
- **Função adicionarAoCarrinho();**

```
function incrementarQuantidadeProduto(idProduto) {  
  idsProdutoCarrinhoComQuantidade[idProduto]++;  
  atualizarPrecoCarrinho(); ←  
  atualizarInformacaoQuantidade(idProduto)  
}
```

Agora vamos exportar a função **atualizarPrecoCarrinho()** para o nosso **arquivo main.js**.

```
export function atualizarPrecoCarrinho() {  
  const precoCarrinho = document.getElementById("preco-total");  
  let precoTotalCarrinho = 0;  
  for (const idProdutoNoCarrinho in idsProdutoCarrinhoComQuantidade) {  
    precoTotalCarrinho += catalogo.find(p => p.id === idProdutoNoCarrinho).preco * idsProdutoCarrinhoComQuantidade[idProdutoNoCarrinho];  
  }  
  precoCarrinho.innerHTML = `Total: ${precoTotalCarrinho}`;  
}
```

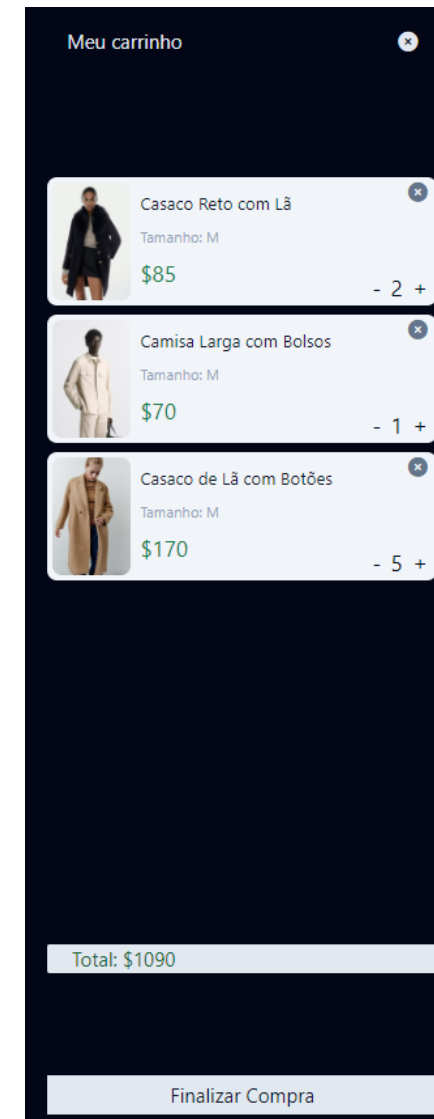
```
JS main.js  
1 import { renderizarCatalogo } from "../src/cartaoProduto";  
2 import { atualizarPrecoCarrinho, inicializarCarrinho } from "../src/menuCarrinho";  
3  
4 renderizarCatalogo();  
5 inicializarCarrinho();  
6 atualizarPrecoCarrinho();
```

Carrinho de compras – Totalizando Preços

Para finalizarmos o nosso Preço total adicionaremos uma estilização. Então vamos retirar o conteúdo de dentro do elemento `<p>` e adicionaremos algumas classes do Tailwind atribuir cor de fundo, cor do texto, margem e vamos deixar o retângulo levemente arredondado.

```
28 <p id="preco-total" class="bg-slate-200 text-green-800 rounded-sm pl-5"></p>
```

E assim construímos essa interface para o nosso carrinho de compras:



Parte 2

LocalStorage



LocalStorage

Compras online nos proporcionam conveniência única. Imagine adicionar produtos ao carrinho, mas adiar a compra. No dia seguinte, ao voltar, surpreendentemente, seus itens ainda estão lá. Essa é a magia do "carrinho de compras persistente", impulsionada pela habilidade do JavaScript e do armazenamento local.

Ele mantém suas escolhas intactas, permitindo que você retome sua jornada de compra exatamente de onde parou, tornando a experiência mais contínua e personalizada. Mas como isso acontece? É aí que entra o JavaScript, a linguagem de programação por trás dessa funcionalidade inteligente, e o **LocalStorage**, uma ferramenta que permite ao navegador armazenar informações.

Quando você adiciona produtos ao carrinho, o JavaScript é acionado. Ele extrai as informações relevantes, como os produtos selecionados e suas quantidades, e utiliza o **LocalStorage** para armazená-las localmente no seu navegador. Esses dados permanecem lá, mesmo após você fechar o site ou reiniciar o navegador.

Quando você retorna ao site, o JavaScript verifica o **LocalStorage** em busca das informações do carrinho. Ele restaura os itens que você adicionou, criando a sensação de que o carrinho "lembra" suas escolhas. Tudo isso acontece de forma discreta, sem exigir que você faça login ou passe por etapas adicionais.

Essa combinação de JavaScript e **LocalStorage** oferece uma experiência de compra notavelmente fluida e personalizada. Sua jornada de compra é preservada, permitindo que você explore e finalize suas escolhas com total liberdade. É um exemplo de como a tecnologia trabalha em segundo plano para tornar sua experiência mais agradável e sem esforço, exatamente como deve ser.

LocalStorage

O **localStorage** é uma funcionalidade oferecida pelo navegador da web que permite que os sites armazenem informações no dispositivo do usuário. Esses dados permanecem disponíveis mesmo após o usuário fechar o navegador ou reiniciar o dispositivo. O **localStorage** é usado para armazenar pequenas quantidades de dados, como configurações do usuário, preferências ou, no contexto do carrinho de compras, informações sobre os produtos selecionados.

Dentro do nosso **arquivo utilidades.js**, vamos adicionar uma função que será responsável por salvar os nossos produtos no **localStorage** e precisaremos exportá-la para conseguir utilizá-la em outras partes do nosso projeto.

```
export function salvarLocalStorage(chave, informacao) {  
  localStorage.setItem(chave, JSON.stringify(informacao));  
}
```

LocalStorage

A função **salvarLocalStorage()** permite que você armazene informações no **localStorage**. Ela usa uma chave para identificar onde os dados serão armazenados e converte o valor da informação em uma string **JSON** antes de armazená-lo.

Vamos analisar detalhadamente essa função:

1- localStorage.setItem(chave, JSON.stringify(informacao)); – Esta linha de código realiza duas ações:

- **localStorage**: É um objeto oferecido pelos navegadores da web que permite que sites armazenem informações localmente no dispositivo do usuário.
- **.setItem(chave, JSON.stringify(informacao))**: Isso chama o método **setItem** do objeto **localStorage**. Esse método permite que você armazene um valor associado a uma chave.

2- chave: É o primeiro argumento passado para a função. É uma string que atua como uma chave para acessar o valor posteriormente. No contexto do **localStorage**, essa chave é usada para identificar onde os dados serão armazenados.

3- informacao: É o segundo argumento passado para a função. Pode ser qualquer tipo de dado que você deseja armazenar. No nosso caso, um objeto.

4- JSON.stringify(informacao): O método **JSON.stringify()** converte o valor da **informacao** em uma string JSON. Isso é necessário porque o **localStorage** só pode armazenar strings.


LocalStorage

Vamos utilizar a função `salvarLocalStorage()` em todos os momentos do arquivo `menuCarrinho.js` em que o nosso objeto/dicionário é atualizado, ou seja vamos importar a função para esse arquivo, e então chamaremos a função `salvarLocalStorage()` dentro das seguintes funções:

- **Função `removerDoCarrinho()`;**
- **Função `incrementarQuantidadeProduto()`;**
- **Função `decrementarQuantidadeProduto()`;**
- **Função `adicionarAoCarrinho()`;**

A sintaxe para executarmos a nossa função `salvarLocalStorage()` será:

```
function removerDoCarrinho(idProduto) {  
  delete idsProdutoCarrinhoComQuantidade[idProduto];  
  salvarLocalStorage("carrinho", idsProdutoCarrinhoComQuantidade);  
  renderizarProdutosCarrinho();  
}
```



A linha de código está essencialmente armazenando os dados contidos em **idsProdutoCarrinhoComQuantidade** no **localStorage** sob a chave "**carrinho**", tornando-os acessíveis para uso posterior.

LocalStorage

Para verificar se a nossa função está funcionando e armazenando as informações dentro do `localStorage`, precisamos acessar o **Dev Tools (ferramentas do desenvolvedor)** dentro do nosso navegador, abaixo está o passo a passo:

- **Abrir o Site:** Abra o navegador da web e acesse o site no qual você deseja trabalhar com o **localStorage**.
- **Abrir as Ferramentas de Desenvolvedor:** Você pode abrir as ferramentas de desenvolvedor de duas maneiras principais:
 - Pressione a tecla **F12**.
 - Clique com o botão direito em qualquer lugar na página e selecione "Inspecionar" no menu de contexto.
- **Navegue até a Aba "Application":** Dentro das ferramentas de desenvolvedor, você verá várias abas, como "Elements", "Console", "Sources", "Network", "Performance", "Memory", entre outras. Procure pela aba "Application" ou "Aplicativos" (o nome pode variar ligeiramente dependendo do navegador).
- **Acesse o LocalStorage:** Dentro da aba "Application", você verá uma seção chamada "Storage" (Armazenamento) no painel esquerdo. Dentro dessa seção, você encontrará "Local Storage". Clique nele para expandir.
- **Visualizar Dados:** Ao clicar em "Local Storage", você verá uma lista de domínios. Clique no domínio do site em que você está interessado. Isso mostrará as chaves e os valores associados que foram armazenados usando o **localStorage**.

LocalStorage

Ao seguir o passo a passo anterior, o seu navegador deve estar parecido com a imagem abaixo (lembrando que para que seja armazenado alguma informação, o seu carrinho deve ter produtos adicionados dentro dele):

The image shows a web application interface for 'ashtag' with a shopping cart overlay. The cart contains five items: 'Casaco de Lã com Botões' (\$170), 'Camisa Larga Acolchoada de Veludo Cotelê' (\$110), 'Casaco Reto com Lã' (\$85), 'Camisa Larga com Bolsos' (\$70), and 'Casaco de Lã com Botões' (\$70). The total is \$1045. The Chrome DevTools Application tab is open, showing the 'Storage' section with 'Local Storage' expanded. The 'carrinho' key is highlighted, showing its value as an array of objects: `[{"1": 5, "2": 1, "5": 4, "6": 1}]`.

Meu carrinho

- Casaco de Lã com Botões
Tamanho: M
\$170 - 1 +
- Camisa Larga Acolchoada de Veludo Cotelê
Tamanho: M
\$110 - 4 +
- Casaco Reto com Lã
Tamanho: M
\$85 - 1 +
- Camisa Larga com Bolsos
Tamanho: M
\$70 - 5 +
- Casaco de Lã com Botões
Tamanho: M
\$70 - 5 +

Total: \$1045

Finalizar Compra

Storage

- Local Storage
 - http://localhost:5173
 - carrinho: [{"1": 5, "2": 1, "5": 4, "6": 1}]

LocalStorage

Agora vamos adicionar uma outra função chamada `lerLocalStorage` no arquivo `utilidades.js`, que será responsável por ler/pegar as informações do `localStorage` e atualizar o nosso objeto.

Analizando a lógica implementada na função, a linha de código: **`return JSON.parse(localStorage.getItem(chave));`** realiza as seguintes ações:

```
export function lerLocalStorage(chave) {  
  return JSON.parse(localStorage.getItem(chave));  
}
```

- **`localStorage.getItem(chave)`**: Isso chama o método **`getItem`** do objeto **`localStorage`**. Esse método permite que você obtenha o valor associado a uma chave especificada. No contexto do código, a chave é passada como argumento.
- **`JSON.parse(...)`**: O método **`JSON.parse()`** é usado para converter uma string JSON em um objeto JavaScript. No contexto do código, ele é usado para converter o valor obtido do **`localStorage`** (que é uma string) em um objeto JavaScript.

A função retorna o objeto JavaScript resultante após converter a string JSON armazenada sob a chave especificada no **`localStorage`**.

Portanto, a função **`lerLocalStorage`** permite que você obtenha e decodifique dados armazenados no **`localStorage`** sob uma determinada chave.

LocalStorage

Vamos importar a função criada para o **arquivo menuCarrinho.js**, e utilizar quando criamos o nosso objeto:

```
src > JS menuCarrinho.js > ...  
1   import { catalogo, lerLocalStorage, salvarLocalStorage } from "../utilidades";  
2  
3   const idsProdutoCarrinhoComQuantidade = lerLocalStorage("carrinho") ?? {};
```

Define a constante **idsProdutoCarrinhoComQuantidade** com o valor recuperado do **localStorage** sob a chave "carrinho", ou um objeto vazio se não houver valor disponível no **localStorage**.

A nossa função irá inicializar ou atualizar o estado do carrinho no código, obtendo os dados previamente armazenados ou criando um objeto vazio se necessário.

- **?? {}** - O operador **??** é o operador de coalescência nula (nullish coalescing operator). Ele é usado para fornecer um valor padrão caso o valor à esquerda seja **null** ou **undefined**. No caso do seu código, se não houver nenhum valor encontrado no **localStorage** (ou seja, se o carrinho ainda não foi armazenado), o operador coalescente nulo retornará um objeto vazio **{}** como valor padrão.

LocalStorage

Para finalizar a implementação do armazenamento das informações do localStorage e a sua inicialização/atualização corretamente. Precisamos exportar a função `renderizarProdutosCarrinho()` do nosso arquivo `menuCarrinho.js` para o arquivo `main.js` (lembrando que este arquivo é responsável por renderizar todas as partes do nosso projeto no site).

```
88  ✓ export function renderizarProdutosCarrinho() {
```

```
JS main.js
1  import { renderizarCatalago } from "../src/cartaoProduto";
2  import { atualizarPrecoCarrinho, inicializarCarrinho, renderizarProdutosCarrinho } from "../src/menuCarrinho";
3
4  renderizarCatalago();
5  inicializarCarrinho();
6  renderizarProdutosCarrinho();
7  atualizarPrecoCarrinho();
```

E com isso o navegador consegue armazenar o nosso carrinho de compras no nosso site E-commerce.

Parte 3

Implementando Filtro Interativo



Implementando Filtro Interativo

Vamos agora adicionar uma funcionalidade especial ao nosso site. Essa funcionalidade permitirá que filtremos os produtos com base em suas características. Faremos isso através de botões que você poderá clicar. Nosso objetivo é separar os produtos em duas categorias: produtos masculinos e produtos femininos.

Para alcançar isso, examinaremos as informações de cada produto. Cada produto possui uma etiqueta associada, uma **"chave"** chamada **"feminino"**. Essa chave possui um **"valor"** que é como uma resposta em código: **"verdadeiro" (true) ou "falso" (false)**. Por exemplo, se o "valor" da chave "feminino" for "verdadeiro", significa que o produto é destinado às mulheres. Se o "valor" for "falso", o produto é para homens.

Essa "chave" e seu "valor" agem como uma conversa codificada com nosso site: se o "valor" da chave "feminino" for "verdadeiro", entendemos que o produto é feminino; se for "falso", é masculino. Nosso site só precisa interpretar essa chave e valor para determinar a categoria do produto.

```
export const catalogo = [
  {
    id: "1",
    marca: 'Zara',
    nome: 'Camisa Larga com Bolsos',
    preco: 70,
    imagem: 'product-1.jpg',
    feminino: false,
  },
  {
    id: "2",
    marca: 'Zara',
    nome: 'Casaco Reto com Lã',
    preco: 85,
    imagem: 'product-2.jpg',
    feminino: true,
  },
]
```

Implementando Filtro Interativo

Dentro do código do arquivo **cartaoProduto.js**, no grupo de classes que cada cartão possui, iremos incluir a chave "feminino" como se fosse uma classe para cada cartão. E após vamos construir a lógica com o operador ternário. O operador ternário é uma forma concisa de escrever condicionais em JavaScript.

Vamos utilizar o operador ternário para determinar a classe a ser atribuída a cada cartão com base na **propriedade "feminino"** do objeto **produtoCatalogo**. Se essa propriedade for definida, o cartão terá a **classe "feminino"**, caso contrário, terá a **classe "masculino"**. Isso nos ajuda a estilizar cada cartão de acordo com seu gênero de produto associado.

```
const cartaoProduto = `<div class="border-solid w-48 m-2 flex flex-col p-2 justify-between shadow-xl shadow-slate-400 rounded-lg group ${produtoCatalogo.feminino ? "feminino" : "masculino"}>
```

Nesse momento precisamos criar um **elemento <section> de id "filtros"** no arquivo index.html que será responsável por encapsular os botões para filtrar o gênero do nosso produto. E vamos utilizar algumas tags que veremos os conceitos a seguir.

Implementando Filtro Interativo

```
<section id="filtros">
  <input id="exibir-todos" type="radio" />
  <label for="exibir-todos">Todos</label>

  <input id="exibir-masculino" type="radio" />
  <label for="exibir-masculino">Masculinos</label>

  <input id="exibir-feminino" type="radio" />
  <label for="exibir-feminino">Femininos</label>
</section>
```

- A tag **<input>** é usada para criar elementos de entrada interativos em um formulário.
- O atributo **id** identifica exclusivamente o elemento, permitindo referências e manipulações específicas usando JavaScript ou CSS.
- O atributo **type** especifica o tipo de input, neste caso, **radio** indica um botão de opção.
- O atributo **type="radio"** indica que é um botão de opção (ou botão de rádio), que permite que o usuário escolha uma opção de um conjunto de opções mutuamente exclusivas.
- A tag **<label>** é usada para associar um rótulo legível a um elemento de entrada, como um botão de opção.
- O atributo **for** está associando o rótulo ao elemento de entrada com o mesmo valor de **id**. Isso cria uma associação entre o rótulo e o botão de opção.
- O texto dentro da tag **<label>** é o rótulo legível que aparecerá na interface para o usuário.

Implementando Filtro Interativo

Vamos aplicar algumas classes do Tailwind para criarmos uma interface mais amigável para os elementos da **<section>** e **<input>**.

Então vamos transforma-lo em containers flexíveis com a **classe flex** e organizaremos melhor os **<input>** na página do navegador:

```
42 <main class="flex flex-col items-center">  
43   <section id="filtros" class="p-10 flex">
```

Agora **cada par de <input> e <label>** irá ficar encapsulado dentro de uma **<div>**, pois iremos aplicar a **classe peer** nos **<input>**, essa classe cria um relacionamento entre elementos irmãos para acionar transições ou animações com base nas interações do usuário.

```
<div class="px-2">  
  <input id="exibir-todos" type="radio" class="peer"/>  
  <label for="exibir-todos">Todos</label>  
</div>  
<div class="px-2">  
  <input id="exibir-masculino" type="radio" class="peer"/>  
  <label for="exibir-masculino">Masculinos</label>  
</div>  
<div class="px-2">  
  <input id="exibir-feminino" type="radio" class="peer"/>  
  <label for="exibir-feminino">Femininos</label>  
</div>
```

Implementando Filtro Interativo

Em cada elemento **<input>**, iremos adicionar a **classe "hidden"** para desativar a interatividade dos textos ("todos", "masculinos", "femininos"), os quais foram ativados com a adição da **classe "peer"**.

Além disso, aplicaremos um estilo para ressaltar visualmente quando esses textos forem clicados. Como padrão, desejamos que a **categoria "Todos"** esteja pré-selecionada para que, ao inicializar nosso site, os produtos já sejam filtrados por padrão, utilizando o **atributo checked**.

```
<div class="px-2">
  <input id="exibir-todos" type="radio" class="peer hidden" checked/>
  <label
    for="exibir-todos"
    class="peer-checked:bg-slate-950 peer-checked:text-slate-200 peer-checked:font-bold p-2 rounded-xl"
    >Todos</label>
</div>
```

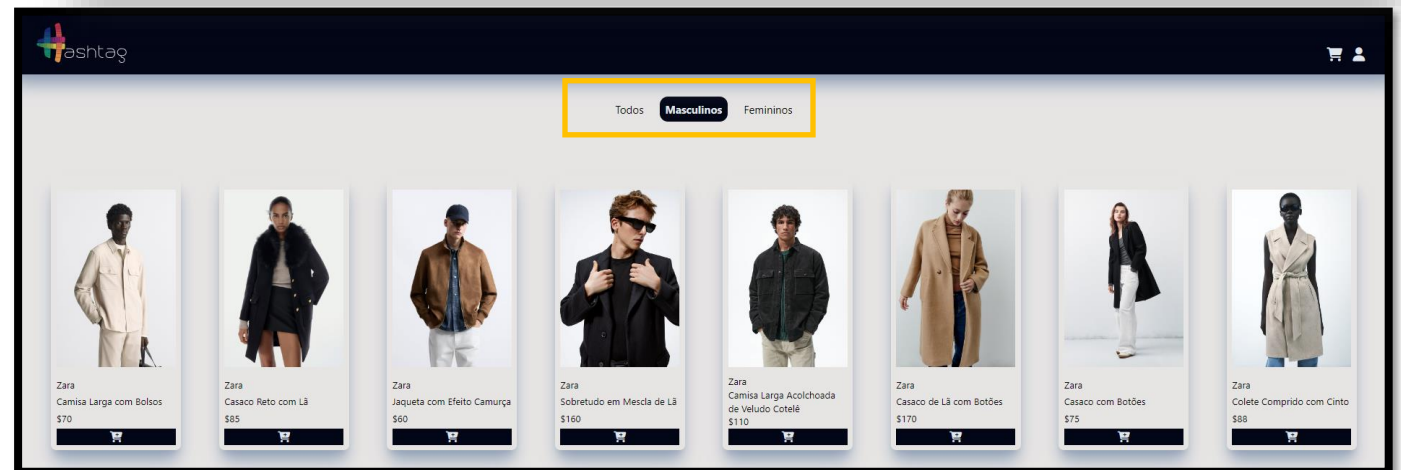
- **peer-checked:bg-slate-950**: Define o fundo do elemento como uma cor específica quando o estado "peer-checked" (interação de clique) está ativado.
- **peer-checked:text-slate-200**: Define a cor do texto como outra cor específica quando o estado "peer-checked" está ativado.
- **peer-checked:font-bold**: Aplica a propriedade de negrito ao texto quando o estado "peer-checked" está ativado.
- **p-2**: Adiciona um espaçamento uniforme ao redor do elemento (padding) com tamanho "2" (moderado).
- **rounded-xl**: Arredonda as bordas do elemento, criando um formato arredondado, sendo "xl" um tamanho maior.

Implementando Filtro Interativo

Para evitar que os botões permaneçam todos selecionados, resultando em todos os três **<input>** marcados simultaneamente, é necessário usar o **atributo "name"**. Isso nos permite selecionar apenas um **<input>** do tipo radio por vez.

E por fim adicionar a **classe "select-none"** nas **<label>**, que remove a capacidade de seleção de texto dentro de um elemento e a **classe "cursor-pointer"**, que altera o cursor do mouse para um ícone de mão quando o mouse passa sobre o elemento, indicando que o elemento é interativo e pode ser clicado.

```
<section id="filtros" class="p-10 flex">
  <div class="px-2">
    <input id="exibir-todos" type="radio" name="filtro" class="peer hidden" checked/>
    <label
      for="exibir-todos"
      class="peer-checked:bg-slate-950 peer-checked:text-slate-200 peer-checked:font-bold p-2 rounded-xl select-none cursor-pointer"
    >Todos</label>
  </div>
  <div class="px-2">
    <input id="exibir-masculino" type="radio" name="filtro" class="peer hidden"/>
    <label
      for="exibir-masculino"
      class="peer-checked:bg-slate-950 peer-checked:text-slate-200 peer-checked:font-bold p-2 rounded-xl select-none cursor-pointer"
    >Masculinos</label>
  </div>
  <div class="px-2">
    <input id="exibir-feminino" type="radio" name="filtro" class="peer hidden"/>
    <label
      for="exibir-feminino"
      class="peer-checked:bg-slate-950 peer-checked:text-slate-200 peer-checked:font-bold p-2 rounded-xl select-none cursor-pointer"
    >Femininos</label>
  </div>
</section>
```



Parte 4

Inteligência dos Filtros



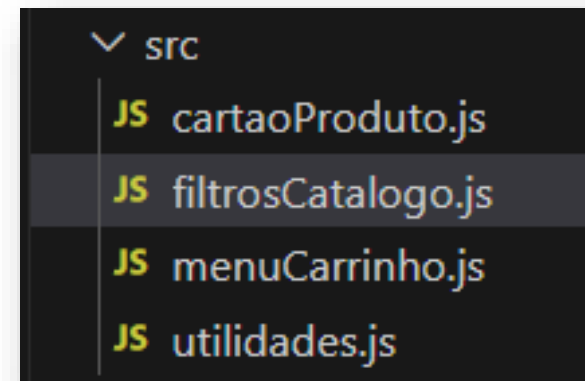
Inteligência dos Filtros

Nesta etapa do projeto, estaremos implementando a lógica dos filtros usando JavaScript. Para começar, criaremos um novo arquivo chamado **"filtrosCatalogo.js"** dentro da pasta **"src"**.

Uma vez que o arquivo for criado, iremos armazenar a referência ao catálogo de produtos em uma variável. A seguir, desenvolveremos a lógica para **três funcionalidades**: exibir todos os produtos, esconder apenas os produtos masculinos e esconder apenas os produtos femininos.

Vamos criar a primeira função chamada **esconderMasculinos()**, que será usada para ocultar todos os produtos masculinos da visualização, tornando-os invisíveis na página.

```
function escondereMasculinos() {  
  const produtosMasculinos = Array.from(  
    catalogoProdutos.getElementsByClassName("masculino")  
  );  
  for (const produto of produtosMasculinos) {  
    produto.classList.add("hidden");  
  }  
}
```



Inteligência dos Filtros

A função **esconderMasculinis()** opera da seguinte maneira:

Ao ser invocada, ela localiza todos os elementos presentes no catálogo de produtos que possuem a classe "masculino". Posteriormente, transforma essa seleção em um array por meio do método **Array.from()**.

A seguir, a função percorre individualmente cada elemento contido no conjunto de produtos masculinos. Durante essa iteração, ela aplica a adição da **classe "hidden"** a cada item. Essa classe, por sua vez, tem o efeito de ocultar o elemento, tornando-o invisível na página.

Agora iremos aplicar a mesma lógica na próxima função que vamos criar, a **esconderFemininos()**, porém recuperaremos a **classe "feminino"**.

```
function escondereFemininos() {  
  const produtosFemininos = Array.from(  
    catalogoProdutos.getElementsByClassName("feminino")  
  );  
  for (const produto of produtosFemininos) {  
    produto.classList.add("hidden");  
  }  
}
```

Inteligência dos Filtros

E por fim, vamos criar A função **exibirTodos()** **que** realiza o seguinte:

1. Ela seleciona todos os elementos no catálogo de produtos que possuem a **classe "hidden"**, ou seja, os produtos que estão atualmente ocultos.
2. Converte essa seleção em um array usando **Array.from()**.
3. Em seguida, itera por cada elemento (produto) do array de produtos ocultos.
4. Para cada produto, remove a **classe "hidden"**, revelando o elemento na página.
5. Essa função é usada para exibir novamente todos os produtos que estavam ocultos anteriormente, removendo a **classe "hidden"** de cada um deles.

```
function exibirTodos() {  
  const produtosEscondidos = Array.from(  
    | catalogoProdutos.getElementsByClassName("hidden")  
  );  
  for (const produto of produtosEscondidos) {  
    | produto.classList.remove("hidden");  
  }  
}
```

Inteligência dos Filtros

Para evitar o problema de produtos que ficam ocultos e não são exibidos no navegador ao clicar em uma categoria, é necessário incorporar a função **exibirTodos()** dentro das funções **esconderMascullinos()** e **esconderFemininos()**. Isso garante que os produtos ocultos sejam exibidos novamente sempre que uma categoria específica for filtrada.

Agora vamos criar a função `inicializarFiltros()` que irá desempenhar o seguinte papel:

- Utilizando **`document.getElementById()`**, a função localiza cada botão de filtro por meio do seu ID único.
- Para o botão "exibir-femininos", ela adiciona um ouvinte de evento que, quando clicado, aciona a função **`esconderMascullinos()`**. Isso resulta na ocultação dos produtos masculinos.
- Da mesma forma, para o botão "exibir-masculinos", é adicionado um ouvinte que, ao ser clicado, executa a função **`esconderFemininos()`**. Isso oculta os produtos femininos.
- Por fim, para o botão "exibir-todos", um ouvinte é acrescentado. Ao ser clicado, a função **`exibirTodos()`** é chamada, restaurando a visualização de todos os produtos.

Essa função é fundamental para permitir que os botões de filtro interajam adequadamente com as funções de filtragem e exibição dos produtos.

Inteligência dos Filtros

Abaixo a estrutura da nossa **função inicializarFiltros()**, ela precisa ser exportada para utilizarmos em outra parte do nosso código.

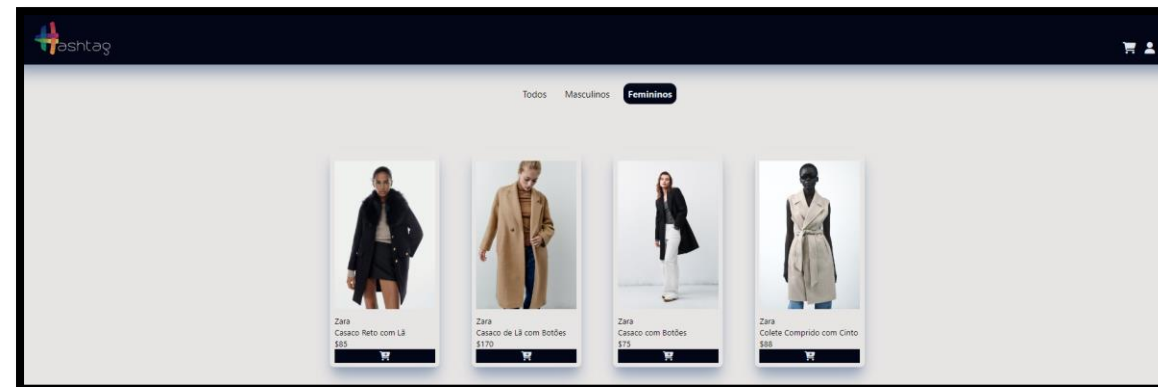
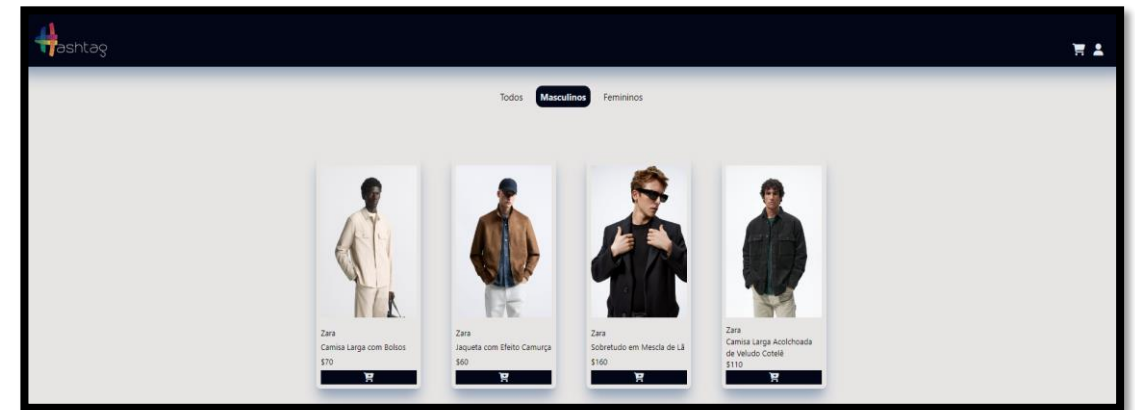
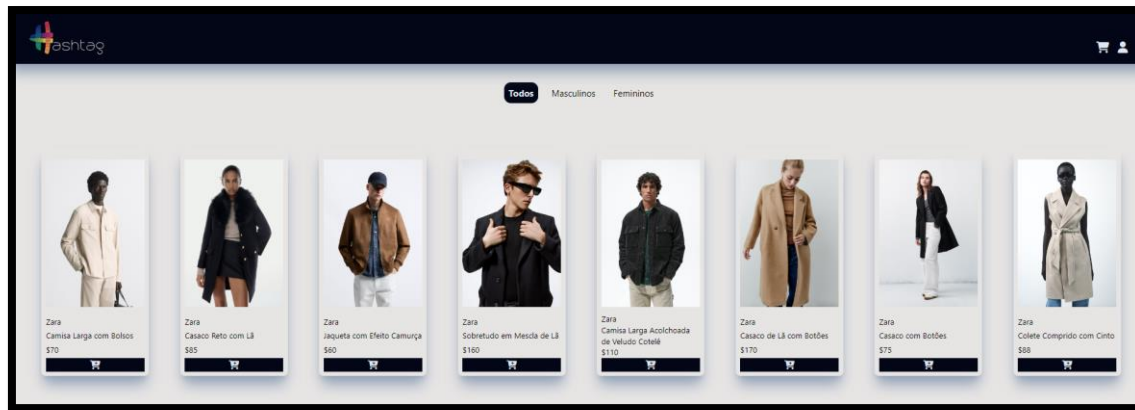
```
export function inicializarFiltros() {  
  document.getElementById("exibir-femininos").addEventListener("click", esconderMasculinis);  
  document.getElementById("exibir-masculinos").addEventListener("click", esconderFemininos);  
  document.getElementById("exibir-todos").addEventListener("click", exibirTodos);  
}
```

Vamos importar a função dentro do **arquivo main.js** e verificar o funcionamento dos nossos filtros.

```
JS main.js  
1 import { renderizarCatalogo } from "../src/cartaoProduto";  
2 import { inicializarFiltros } from "../src/filtrosCatalogo";  
3 import { atualizarPrecoCarrinho, inicializarCarrinho, renderizarProdutosCarrinho } from "../src/menuCarrinho";  
4  
5 renderizarCatalogo();  
6 inicializarCarrinho();  
7 inicializarFiltros();  
8 renderizarProdutosCarrinho();  
9 atualizarPrecoCarrinho();
```

Inteligência dos Filtros

Assim, ao selecionarmos os filtros em nosso site, podemos observar a funcionalidade aplicada e operando de maneira precisa e eficaz.



Parte 5

Checkout



Checkout

Nesta fase do projeto, embarcaremos na construção da nossa página de checkout. Iremos estabelecer sua estrutura, conferir-lhe estilo e infundir inteligência por meio do JavaScript.

Iremos inserir duas classes no botão "Finalizar Compra", a fim de implementar um efeito de destaque que será ativado quando o cursor do mouse for movido sobre o botão.

```
<button class="bg-slate-200 text-slate-900 p-1 hover:text-slate-200 hover:bg-slate-900">Finalizar Compra</button>
```

Esse botão será responsável por nos encaminhar até a página de checkout, então antes de implementarmos essa funcionalidade vamos construir o checkout. Para isso vamos criar um novo **arquivo html** na raiz do nosso projeto chamado **checkout.html**, que terá a estrutura dessa nossa página. A estrutura inicial dessa página será:

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Título da página</title>
7   <link rel="stylesheet" href="style.css">
8   <link
9     rel="stylesheet"
10    href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.2/css/all.min.css"
11    integrity="sha512-z3glpd7yknf1YoNbCzqRKc4qyor8gaKU1qmn+CShxbuBusANI9QpRohGBreCFkKxLhe16S9CQXFEbbKuqLg0DA=="
12    crossorigin="anonymous"
13    referrerpolicy="no-referrer"
14  />
15 </head>
16 <body class="bg-stone-200 flex flex-col">
17   <header class="flex text-xl bg-slate-950 px-8 py-4 justify-between sticky top-0 shadow-xl shadow-slate-400 z-10">
18     
23   </header>
24   <form class="flex"></form>
25 </body>
26 </html>
```

Checkout

Observe que adicionamos uma tag nova chamada "form". A tag **<form>** é uma marcação HTML usada para criar um formulário em uma página web. Formulários são elementos interativos que permitem aos usuários inserir e enviar dados para um servidor web. A tag **<form>** envolve campos de entrada, botões de envio e outros elementos relacionados a coletar informações do usuário.

A tag **<form>** oferece um mecanismo estruturado para coletar informações dos usuários e transmiti-las para processamento, validação ou armazenamento em um servidor web.

Vamos dividir o nosso formulário em 3 sessões: Dados do usuário, Dados de pagamento e exibir os produtos que o usuário adicionou no carrinho de compras.

```
<form class="flex">  
  <section id="dados-usuario"></section>  
  <section id="dados-pagamento"></section>  
  <section id="produtos-checkout"></section>  
</form>
```

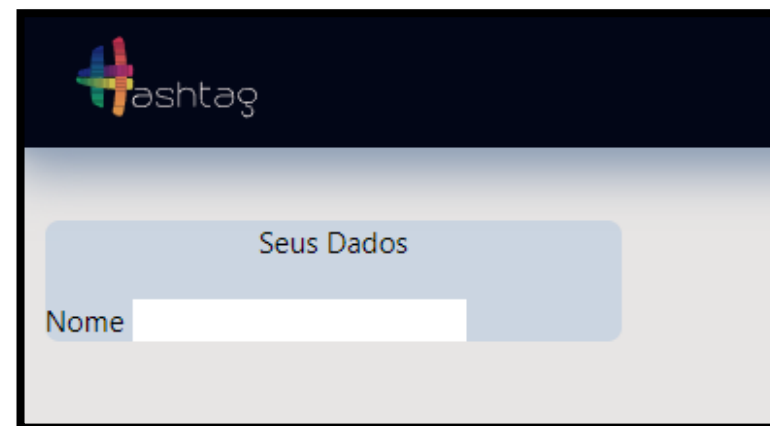
Checkout

Antes de adicionarmos conteúdos aos nossos elementos vamos adicionar algumas classes do Tailwind para organizarmos cada elemento na página.

```
<form class="flex w-100 px-5 gap-4">
  <section id="dados-usuario" class="w-1/3"></section>
  <section id="dados-pagamento" class="w-1/3"></section>
  <section id="produtos-checkout" class="w-1/3"></section>
</form>
```

Vamos começar a trabalhar na primeira sessão, que será os Dados do usuário. Então vamos adicionar o texto "Seus Dados" e criar um elemento `<input>` do tipo texto e uma `<label>` para nomearmos esse `<input>`.

```
<form class="flex w-100 px-5 gap-4 py-10">
  <section id="dados-usuario" class="w-1/3 bg-slate-300 rounded-lg">
    <p class="mb-5 text-center">Seus Dados</p>
    <div>
      <label for="nome">Nome</label>
      <input type="text" id="nome" />
    </div>
  </section>
  <section id="dados-pagamento" class="w-1/3"></section>
  <section id="produtos-checkout" class="w-1/3"></section>
</form>
```



Checkout

Vamos ver algumas classes do Tailwind que estamos utilizando para estilizar e estruturar a nossa sessão Dados do usuário.

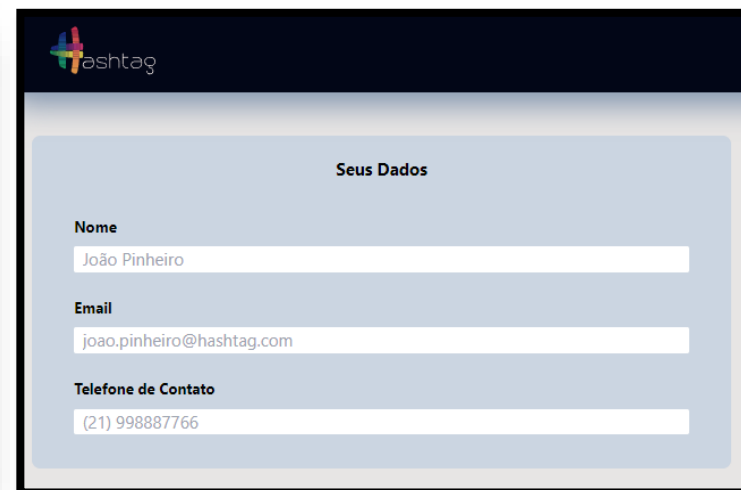
- **flex**: Aplica o estilo de flexbox para criar um layout flexível.
- **w-100**: Define a largura como 100% do contêiner pai.
- **px-n**: Adiciona um espaçamento de preenchimento horizontal de "n" unidades.
- **gap-n**: Define um espaçamento entre os elementos filho de "n" unidades.
- **py-n**: Adiciona um espaçamento de preenchimento vertical de "n" unidades.
- **grow**: Provavelmente é uma classe personalizada que ajusta algum comportamento de crescimento do elemento.
- **w-1/3**: Define a largura como um terço do contêiner pai.
- **bg-white**: Define o fundo como branco.
- **rounded-lg**: Aplica bordas arredondadas com um raio específico.

- **mb-n**: Adiciona margem inferior de "n" unidades.
- **text-center**: Centraliza o texto horizontalmente.
- **font-bold**: Aplica uma formatação de fonte em negrito.
- **flex-col**: Define o layout flexível como coluna (itens empilhados verticalmente).
- **text-sm**: Define o tamanho do texto como pequeno.
- **border-n**: Adiciona uma borda de "n" unidades.
- **border-slate-200**: Define a cor da borda como um tom de cinza.
- **type="text", type="email", type="tel"**: Define os tipos de entrada para os campos de texto, email e telefone, respectivamente.
- **placeholder="conteúdo"**: Define um texto de placeholder que aparece nos campos de entrada quando eles estão vazios.
- **required**: Indica que os campos de entrada são obrigatórios e não podem ser enviados vazios.

Checkout

Para completarmos a sessão Dados do usuário, vamos adicionar os demais campos que serão de email e telefone de contato , e eles terão tipos de input de acordo com o tipo de dados que esperamos nesses campos.

```
<section id="dados-usuario" class="w-1/3 bg-slate-300 rounded-lg py-5">
  <p class="mb-5 text-center font-bold">Seus Dados</p>
  <div class="flex flex-col px-10 py-3">
    <label for="nome" class="font-bold text-sm mb-2">Nome</label>
    <input type="text" id="nome" class="rounded-sm px-2" placeholder="João Pinheiro" required/>
  </div>
  <div class="flex flex-col px-10 py-3">
    <label for="email" class="font-bold text-sm mb-2">Email</label>
    <input type="email" id="email" class="rounded-sm px-2" placeholder="joao.pinheiro@hashtag.com" required/>
  </div>
  <div class="flex flex-col px-10 py-3">
    <label for="telefone" class="font-bold text-sm mb-2">Telefone de Contato</label>
    <input type="tel" id="telefone" class="rounded-sm px-2" placeholder="(21) 998887766 " required/>
  </div>
</section>
```



The image shows a web form titled "Seus Dados" (Your Data) with a dark blue header containing the "hashtag" logo. The form is a light blue rounded rectangle with three sections: "Nome" (Name) with a text input field containing "João Pinheiro"; "Email" with an email input field containing "joao.pinheiro@hashtag.com"; and "Telefone de Contato" (Contact Phone) with a telephone input field containing "(21) 998887766".

Para o usuário o preenchimento desses campos devem ser obrigatórios e para isso vamos utilizar a **propriedade "required"** que garante isso.

Checkout

Vamos aplicar mais algumas estilização para melhorarmos ainda mais a interface da página de checkout. O `<body>` irá receber uma classe "h-screen" para preencher toda a tela do navegador, enquanto o nosso formulário terá a classe "grow" que fará ela obter o tamanho adequado de acordo com a tela, respeitando os espaçamentos já implementados.

```
<body class="bg-stone-200 flex flex-col h-screen">
```

Nesse momento a estrutura da nossa primeira sessão ficou dessa forma:

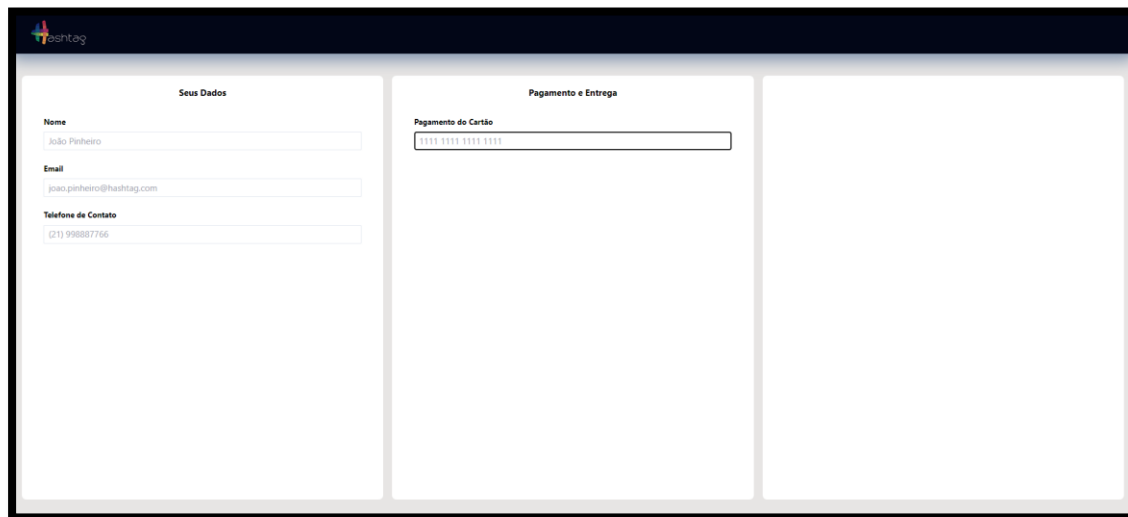
```
<form class="flex w-100 px-5 gap-4 py-10 grow">
  <section id="dados-usuario" class="w-1/3 bg-white rounded-lg py-5">
    <p class="mb-5 text-center font-bold">Seus Dados</p>
    <div class="flex flex-col px-10 py-3">
      <label for="nome" class="font-bold text-sm mb-2">Nome</label>
      <input type="text" id="nome" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="João Pinheiro" required/>
    </div>
    <div class="flex flex-col px-10 py-3">
      <label for="email" class="font-bold text-sm mb-2">Email</label>
      <input type="email" id="email" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="joao.pinheiro@hashtag.com" required/>
    </div>
    <div class="flex flex-col px-10 py-3">
      <label for="telefone" class="font-bold text-sm mb-2">Telefone de Contato</label>
      <input type="tel" id="telefone" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="(21) 998887766 " required/>
    </div>
  </section>
```

Checkout

Agora vamos construir a segunda sessão que será **"Pagamento e Entrega"**, o id dessa sessão vai ser alterado para "pagamento-entrega", e sua estrutura e estilo iniciará no mesmo formado que a sessão Dados do usuário, porque queremos padronizar o nosso site. Assim como a terceira sessão receberá a mesma estrutura de estilos.

```
<section id="pagamento-entrega" class="w-1/3 bg-white rounded-lg py-5">
  <p class="mb-5 text-center font-bold">Pagamento e Entrega</p>
  <div class="flex flex-col px-10 py-3">
    <label for="numero-cartao" class="font-bold text-sm mb-2">Pagamento do Cartão</label>
    <input type="text" id="numero-cartao" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="1111 1111 1111 1111" required/>
  </div>
```

A estrutura do site deve estar parecida com a imagem abaixo:



The image shows a mockup of a checkout page layout. It features a dark blue header with the 'hashtag' logo. Below the header, the page is divided into three vertical columns. The first column, titled 'Seus Dados', contains three input fields: 'Nome' (filled with 'João Pinheiro'), 'Email' (filled with 'joao.pinheiro@hashtag.com'), and 'Telefone de Contato' (filled with '(21) 998887766'). The second column, titled 'Pagamento e Entrega', contains a single input field for 'Pagamento do Cartão' with the placeholder '1111 1111 1111 1111'. The third column is currently empty.

Checkout

Para os próximos elementos da sessão "Pagamento e Entrega", vamos envolver dois campos por um elemento **<div>** que possui a **classe flex**, e adicionaremos as estruturas das informações '**CVV**' e '**Data de validade**'.

```
<div class="flex">
  <div class="flex flex-col px-10 py-3 w-1/2">
    <label for="cvv" class="font-bold text-sm mb-2">CVV</label>
    <input type="text" id="cvv" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="222" required/>
  </div>
  <div class="flex flex-col px-10 py-3 w-1/2">
    <label for="data-expiracao" class="font-bold text-sm mb-2">Data de Validade</label>
    <input type="text" id="data-expiracao" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="10/24" required/>
  </div>
</div>
```

E os próximos campos seguiram a estrutura inicial das nossas sessões com as informações de CEP e endereço de entrega.

```
<div class="flex flex-col px-10 py-3">
  <label for="cep" class="font-bold text-sm mb-2">CEP</label>
  <input type="text" id="cep" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="11122-333" required/>
</div>
<div class="flex flex-col px-10 py-3">
  <label for="endereco" class="font-bold text-sm mb-2">Endereço</label>
  <input type="text" id="endereco" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="Rua que sobe e desce" required/>
</div>
```


Checkout

E a estrutura das informações de número da residência e o complemento seguirá o padrão dos elementos "CVV" e "Data de validade":

```
<div class="flex">  
  <div class="flex flex-col px-10 py-3 w-1/2">  
    <label for="numero" class="font-bold text-sm mb-2">Número</label>  
    <input type="text" id="numero" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="444" required/>  
  </div>  
  <div class="flex flex-col px-10 py-3 w-1/2">  
    <label for="complemento" class="font-bold text-sm mb-2">Complemento</label>  
    <input type="text" id="complemento" class="rounded-sm px-2 border-2 border-slate-200 py-1" placeholder="555" required/>  
  </div>  
</div>
```

Com essas novas implementações a estrutura da página de checkout está com a interface da imagem abaixo:

The screenshot shows a checkout form for 'hashtag' with a dark blue header. The form is divided into three columns. The first column, titled 'Seus Dados', contains fields for Name (João Pinheiro), Email (joao.pinheiro@hashtag.com), and Phone (21) 998867766. The second column, titled 'Pagamento e Entrega', contains fields for Card Number (1111 1111 1111 1111), CVV (222), Card Validity Date (10/24), CEP (11122-333), Address (Rua que sobe e desce), Number (444), and Complement (555). The third column is empty.

Checkout

Agora avançaremos para a última seção da página de checkout, onde exibimos os produtos adicionados pelo usuário ao carrinho de compras. Iremos copiar o código responsável por renderizar esses produtos do arquivo **menuCarrinho.js** e colá-lo no arquivo **utilidades.js**. Entretanto, removeremos quaisquer funcionalidades que não sejam relevantes para essa seção específica do checkout.

Vamos renomear a função para **"desenharProdutoCarrinhoSimples"** e incluir dois novos parâmetros, **"idContainerHTML"** e **"quantidadeProduto"**. Precisamos dessa adaptação para usar a função em diversos locais do projeto. Além disso, removeremos todos os botões do interior da função, já que não desejamos as funcionalidades de adicionar ou remover produtos nesta página.

```
src > JS utilidades.js > ...
76 export function desenharProdutoCarrinhoSimples(idProduto, idContainerHTML, quantidadeProduto) {
77   const produto = catalogo.find((p) => p.id === idProduto);
78   const containerProdutosCarrinho = document.getElementById(idContainerHTML);
79
80   const elementoArticle = document.createElement("article"); //cria a tag <article></article>
81   const articleClasses = ["flex", "bg-stone-200", "rounded-lg", "p-1", "relative"];
82   for (const articleClass of articleClasses) {
83     elementoArticle.classList.add(articleClass);
84   } // adiciona as classes <article class="flex bg-slate-100 rounded-lg p-1 relative"></article>
85
86   const cartaoProdutoCarrinho = `
87     
92     <div class="p-2 flex flex-col justify-between">
93       <p class="text-slate-900 text-sm">${produto.nome}</p>
94       <p class="text-slate-400 text-xs">Tamanho: M</p>
95       <p class="text-green-700 text-lg">${produto.preco}</p>
96     </div>
97     <div class="flex text-slate-950 items-end absolute bottom-0 right-2 text-lg">
98
99       <p id="quantidade-${produto.id}" class="ml-2">${quantidadeProduto}</p>
100
101     </div>`;
102
103   elementoArticle.innerHTML = cartaoProdutoCarrinho;
104   containerProdutosCarrinho.appendChild(elementoArticle);
105 }
```

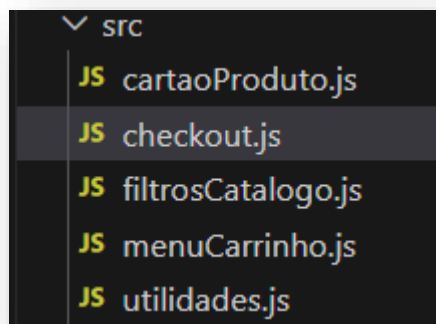
Checkout

No arquivo **checkout.html**, vamos adicionar dois novos elementos **<section>** dentro da terceira sessão. Cada uma dessas seções receberá um **ID**, sendo um "**container-produtos-checkout**" e o outro "**celula-checkout**". A razão para essa divisão é separar a exibição dos cartões dos produtos em uma parte e a apresentação do preço final do carrinho, juntamente com o botão para concluir a compra, em outra parte.

Essa organização ajudará a criar uma divisão clara entre as informações dos produtos e as informações de pagamento/finalização da compra na página de checkout.

```
<section id="produtos-checkout" class="w-1/3 bg-white rounded-lg py-5">
  <section id="container-produtos-checkout"></section>
  <section id="celula-checkout"></section>
</section>
```

Conforme a boa prática, vamos criar um arquivo chamado **checkout.js**, dentro da pasta "src", que será encarregado de conter a lógica da nossa página de checkout.



Checkout

Vamos exportar a função '**desenharProdutoCarrinhoSimples**' que criamos no arquivo **utilidades.js**, para que possamos importá-la no recém-criado **checkout.js**.

Definiremos uma função chamada **desenharProdutosCheckout** que é responsável por desenhar os produtos no carrinho na página de checkout.

```
src > JS checkout.js > ...
1  import { desenharProdutoCarrinhoSimples, lerLocalStorage } from "../utilidades";
2
3  function desenharProdutosCheckout() {
4      const idsProdutoCarrinhoComQuantidade = lerLocalStorage("carrinho");
5      for (const idProduto in idsProdutoCarrinhoComQuantidade) {
6          desenharProdutoCarrinhoSimples(idProduto, "container-produtos-checkout", idsProdutoCarrinhoComQuantidade[idProduto]);
7      }
8  }
9
10 desenharProdutosCheckout();
```

- **const idsProdutoCarrinhoComQuantidade = lerLocalStorage("carrinho");**: Aqui, é recuperado um objeto do Local Storage chamado "carrinho", que contém os IDs dos produtos presentes no carrinho, juntamente com as quantidades de cada produto.
- **for (const idProduto in idsProdutoCarrinhoComQuantidade) { ... }**: Isso inicia um loop que percorre cada ID de produto presente no objeto **idsProdutoCarrinhoComQuantidade**.
- **desenharProdutoCarrinhoSimples(idProduto,"container-produtos-checkout",idsProdutoCarrinhoComQuantidade[idProduto]);**:

Chama a função **desenharProdutoCarrinhoSimples** com três argumentos: o ID do produto, o ID do contêiner HTML onde o produto será desenhado ("container-produtos-checkout") e a quantidade desse produto no carrinho.

Checkout

E nesse momento vamos adicionar alguns estilos para que a estrutura da página de checkout fique da seguinte maneira:

```
const articleClasses = ["flex", "bg-stone-200", "rounded-lg", "p-1", "relative", "mb-2"];
```

The screenshot shows a checkout form for a store named 'hashtag'. The form is divided into three main sections:

- Seus Dados (Your Data):**
 - Nome: João Pinheiro
 - Email: joao.pinheiro@hashtag.com
 - Telefone de Contato: (21) 998887766
- Pagamento e Entrega (Payment and Delivery):**
 - Pagamento do Cartão:** 1111 1111 1111 1111
 - CVV:** 222
 - Data de Validade:** 10/24
 - CEP:** 11122-333
 - Endereço:** Rua que sobe e desce
 - Número:** 444
 - Complemento:** 555
- Cartão de Produtos:**
 - Camisa Larga com Bolsos (Tamanho: M) - \$70 - 5
 - Casaco Reto com Lã (Tamanho: M) - \$85 - 1
 - Camisa Larga Acolchoada de Veludo Cotelê (Tamanho: M) - \$110 - 4
 - Casaco de Lã com Botões (Tamanho: M) - \$170 - 1

O próximo passo é adicionar o preço total e criar nosso botão de finalizar compras e sua funcionalidade.

Checkout

A estrutura implementada para a nossa sessão exibir os produtos na página de checkout, terá o seguinte código já aplicando todas as estilizações que vimos até o momento do tailwind.

```
<section id="celula-checkout" class="flex flex-col px-2 h-1/5 justify-around">
  <p id="preco-total" class="bg-slate-200 text-green-800 rounded-lg pl-5">Total: $200</p>
  <button class="bg-slate-500 text-slate-200 p-1 hover:bg-slate-300 " type="submit">Finalizar Compra</button>
</section>
```

O atributo **type** está definido como **"submit"**. Isso indica que, quando o botão for pressionado dentro de um formulário, ele enviará os dados do formulário para processamento. Essa é uma funcionalidade comum para botões em formulários que permitem que os dados inseridos pelo usuário sejam enviados para algum destino, como um servidor, para processamento adicional.

The screenshot displays a checkout interface for a store named 'hashtag'. It is divided into three main sections:

- Seus Dados (Your Data):** Contains input fields for Name (João Pinheiro), Email (joao.pinheiro@hashtag.com), Phone (21) 998887766, and Address (Rua que sobe e desce).
- Pagamento e Entrega (Payment and Delivery):** Contains input fields for Card Number (1111 1111 1111 1111), CVV (222), Expiry Date (10/24), CEP (11122-333), and Complement (555).
- Items:** A list of items with their names, prices, and quantities:
 - Camisa Larga com Botões: \$70, 5
 - Casaco Reto com Lã: \$85, 1
 - Camisa Larga Acolchoada de Veludo Cotelê: \$110, 4
 - Casaco de Lã com Botões: \$170, 1

At the bottom, there is a summary bar showing 'Total: \$200' and a 'Finalizar Compra' button.

Checkout

A próxima etapa envolve estabelecer a ligação entre a nossa página principal e a página de checkout. Essa conexão será ativada quando o usuário clicar no botão "Finalizar Compra" localizado no carrinho de compras. A primeira verificação que precisamos realizar é se o carrinho possui produtos adicionados.

Para isso, vamos criar uma função chamada **"irParaCheckout()"** dentro do arquivo **menuCarrinho.js**, visto que estamos tratando de uma funcionalidade necessária para o botão localizado dentro do carrinho de compras.

```
function irParaCheckout() {  
  if(Object.keys(idsProdutoCarrinhoComQuantidade).length === 0) {  
    return;  
  }  
}
```

A parte **Object.keys** é usada para obter todas as chaves (IDs dos produtos) do objeto **idsProdutoCarrinhoComQuantidade**. **idsProdutoCarrinhoComQuantidade.length**: Isso retorna o número de elementos no array resultante de **Object.keys(idsProdutoCarrinhoComQuantidade)**.

- **=== 0**: Isso verifica se o comprimento (ou seja, a quantidade de produtos no carrinho) é igual a zero.
- **return**: Se o comprimento for zero (ou seja, o carrinho está vazio), a função retorna imediatamente, sem realizar nenhuma ação adicional.

Checkout

Agora vamos criar a lógica para redirecionar o usuário da página principal para a página checkout:

```
function irParaCheckout() {  
  if(Object.keys(idsProdutoCarrinhoComQuantidade).length === 0) {  
    return;  
  }  
  window.location.href = window.location.origin + "/checkout.html";  
}
```

- **window.location.href**: Isso se refere à propriedade **href** do objeto **window.location**, que contém a URL completa da página atual.
- **window.location.origin**: Essa parte pega o "origem" da URL da página atual, ou seja, o protocolo (como "http" ou "https"), o domínio e a porta (se houver).
- **+ "/checkout.html"**: Isso adiciona a parte "/checkout.html" à origem, criando assim a URL completa da página de checkout.

Portanto, a função redireciona o usuário para a página de checkout definindo o **window.location.href** para a URL completa da página de checkout. O usuário será levado para a página **checkout.html**.

Checkout

Para incorporar a funcionalidade que criamos, vamos seguir os seguintes passos:

No arquivo **index.html**, dentro do elemento **<button>** "Finalizar Compra", vamos adicionar um atributo **id** com o valor "finalizar-compra".

```
<button  
  id="finalizar-compra"  
  class="bg-slate-200 text-slate-900 p-1 hover:text-slate-200 hover:bg-slate-900"  
>Finalizar Compra</button>
```

Agora, no arquivo **menuCarrinho.js**, dentro da função **inicializarCarrinho()**, vamos adicionar a chamada da função **irParaCheckout()** como manipuladora do evento de clique para o botão "Finalizar Compra". Para isso, vamos recuperar o botão pelo ID e adicionar um ouvinte de evento de clique.

```
export function inicializarCarrinho() {  
  const botaoFecharCarrinho = document.getElementById("fechar-carrinho");  
  const botaoAbrirCarrinho = document.getElementById("abrir-carrinho");  
  const botaoIrParaCheckout = document.getElementById("finalizar-compra")  
  
  botaoFecharCarrinho.addEventListener("click", fecharCarrinho);  
  botaoAbrirCarrinho.addEventListener("click", abrirCarrinho);  
  botaoIrParaCheckout.addEventListener("click", irParaCheckout);  
}
```

Dessa forma, associamos a função **irParaCheckout()** ao botão "Finalizar Compra" através do evento de clique. Quando o carrinho de compras tiver produtos e o usuário clicar no botão, ele será redirecionado para a página de checkout.

Parte 6

Finalizar Compra



Finalizar Compra

Na pasta principal do nosso projeto, vamos criar a terceira página chamada **pedidos.html**. Essa página terá a função de armazenar o histórico de compras do usuário, exibir a confirmação de que o pedido foi realizado com sucesso e efetuar o esvaziamento do carrinho de compras. Quando o usuário concluir a compra na página de checkout, ele será automaticamente redirecionado para esta página.

A base da nossa página de pedido tem a seguinte estrutura:

```
<> pedidos.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Título da página</title>
7    <link rel="stylesheet" href="style.css">
8    <link
9      rel="stylesheet"
10     href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.4.2/css/all.min.css"
11     integrity="sha512-z3gLpd7yknf1YoNbCzqRKc4qyor8gaKU1qmn+CSHxbuBusANI9QpRohGBreCFkKxLhei6S9CQXFEbbKuqLg0DA=="
12     crossorigin="anonymous"
13     referrerpolicy="no-referrer"
14   />
15 </head>
16 <body class="bg-stone-200 flex flex-col h-screen">
17   <header class="flex text-xl bg-slate-950 px-8 py-4 justify-between sticky top-0 shadow-xl shadow-slate-400 z-10">
18     
23   </header>
24 </body>
25 </html>
```

Finalizar Compra

Agora que temos a base da nossa página Pedidos, podemos construir a lógica para o botão "Finalizar Compra" da página de checkout, lembrando que ao clicar nesse botão o usuário será redirecionado para a página de Pedidos.

Então dentro do arquivo checkout.js, definiremos uma função finalizarCompra(), é uma função de evento que lida com o processo de finalização da compra quando o botão "Finalizar Compra" é clicado. A seguinte lógica será implementada:

A função é responsável por prevenir o comportamento padrão do botão de envio, impedindo a recarga da página, e em seguida, redireciona o usuário para a página de pedidos. Isso ocorre quando o botão "Finalizar Compra" é clicado na página de checkout.

```
function finalizarCompra(evento) {  
    evento.preventDefault();  
    window.location.href = window.location.origin + "/pedidos.html"  
}
```

Finalizar Compra

Vamos entender os conceitos aplicados dentro dela:

- **evento**: Isso representa o objeto de evento gerado quando o botão é clicado. É passado como parâmetro para a função.
- **evento.preventDefault()**: Essa linha de código impede o comportamento padrão do botão (que é enviar o formulário ou recarregar a página). Isso é feito usando o método **preventDefault()** para evitar que a página seja recarregada.
- **window.location.href**: Isso se refere à propriedade **href** do objeto **window.location**, que contém a URL completa da página atual.
- **window.location.origin**: Essa parte pega o "origem" da URL da página atual, ou seja, o protocolo (como "http" ou "https"), o domínio e a porta (se houver).
- **+ "/pedidos.html"**: Isso adiciona a parte **"/pedidos.html"** à origem, criando assim a URL completa da página de pedidos.
- O resultado é que a linha **window.location.href = window.location.origin + "/pedidos.html";** redireciona o usuário para a página de pedidos após a finalização da compra.

Finalizar Compra

Com a nossa função criada, vamos adicionar um evento "submit" a ela, que fará com que a função seja executada, para isso vamos construir a seguinte linha de código:

```
document.addEventListener("submit", (evt) => finalizarCompra(evt));
```

Essa linha de código adiciona um ouvinte de evento ao documento (a página da web) que está aguardando o evento de envio de formulário (**submit**). Vou detalhar cada parte:

- **document**: Isso se refere ao objeto **document**, que representa o documento HTML carregado na página da web.
- **.addEventListener("submit", (evt) => finalizarCompra(evt))**: Isso adiciona um ouvinte de evento ao documento. O evento específico é o evento de envio de formulário (**submit**), que ocorre quando um formulário é enviado.
- **(evt) => finalizarCompra(evt)**: Aqui, estamos passando uma função como o segundo argumento do método **addEventListener**. Essa função será executada sempre que o evento de envio de formulário for acionado. O parâmetro **evt** representa o objeto de evento gerado quando o evento ocorre.
- **finalizarCompra(evt)**: Dentro da função, estamos chamando a função **finalizarCompra** passando o objeto de evento **evt** como argumento. Isso permite que a função **finalizarCompra** seja executada quando o evento de envio de formulário acontecer.

Finalizar Compra

No Javascript existe uma função chamada **"new Date()"**, ela irá criar um objeto que representa a data e a hora atuais. Isso significa que o objeto **Date** terá informações sobre o dia, mês, ano, hora, minuto, segundo e milissegundos no exato momento em que o código é executado.

E vamos adiciona-la dentro da nossa função **"finalizarCompra()"** porque queremos armazenar essa informação. Vamos usar novamente as estruturas da função **"irParaCheckout()"**, como a **verificação do localStorage**, uso do **conceito Object.keys()** para continuar a nossa implementação na função **"finalizarCompra()"**.

Além disso, iremos implementar dentro do **arquivo utilidades.js**, uma função para apagar o conteúdo do localStorage (limpar o carrinho), que será utilizada dentro da função **"finalizarCompra()"**. A função **apagarDoLocalStorage** permite que você remova um item específico do armazenamento local do navegador, com base na chave fornecida como argumento. Essa função será importada e usada em outros lugares do código.

```
export function apagarDoLocalStorage(chave) {  
  | localStorage.removeItem(chave);  
  |  
}
```

Finalizar Compra

Vamos adicionar mais funcionalidades à nossa função **finalizarCompra()** para permitir que o histórico de pedidos seja salvo no armazenamento local após a conclusão da compra. Agora, vamos explorar o restante da lógica que está sendo aplicada nessa função.

```
function finalizarCompra(evento) {
  evento.preventDefault();
  const idsProdutoCarrinhoComQuantidade = lerLocalStorage("carrinho") ?? {};
  if(Object.keys(idsProdutoCarrinhoComQuantidade).length === 0) {
    return;
  };

  const dataAtual = new Date();
  const pedidoFeito = {
    dataPedido: dataAtual,
    pedido: idsProdutoCarrinhoComQuantidade
  }

  const historicoDePedidos = lerLocalStorage("historico") ?? [];
  const historicoDePedidosAtualizados = [pedidoFeito, ...historicoDePedidos];

  salvarLocalStorage("historico", historicoDePedidosAtualizados);
  apagarDoLocalStorage("carrinho");

  window.location.href = window.location.origin + "/pedidos.html"
}
```

const pedidoFeito = { ... }: Aqui, um objeto chamado **pedidoFeito** está sendo criado. Ele possui duas propriedades:

- **dataPedido:** Essa propriedade recebe o valor da variável **dataAtual**, que é a data e hora atuais.
- **pedido:** Essa propriedade recebe o valor da variável **idsProdutoCarrinhoComQuantidade**, que provavelmente contém os IDs dos produtos e suas quantidades no carrinho.

Finalizar Compra

- **const historicoDePedidos = lerLocalStorage("histórico") ?? [];** Aqui, a função **lerLocalStorage** é usada para recuperar o histórico de pedidos do armazenamento local. Se não houver histórico, um array vazio será usado como valor padrão.
- **const historicoDePedidosAtualizados = [pedidoFeito, ...historicoDePedidos];** Nesta linha, o novo pedido (definido como **pedidoFeito**) é adicionado ao início do histórico de pedidos existente, criando um array **historicoDePedidosAtualizados**.
- **salvarLocalStorage("historico", historicoDePedidosAtualizados);** Essa linha usa a função **salvarLocalStorage** para armazenar o array **historicoDePedidosAtualizados** no armazenamento local com a chave "historico".
- **apagarDoLocalStorage("carrinho");** Aqui, a função **apagarDoLocalStorage** é usada para remover um item com a chave "carrinho" do armazenamento local.

Parte 7

Página de Pedidos



Página de Pedidos

Agora vamos construir a estrutura da nossa página de pedidos. Para isso, criaremos um arquivo chamado **pedidos.js**, que conterá toda a lógica para essa página.

Vamos importar as funções **lerLocalStorage** e **desenharProdutoCarrinhoSimples**, e então criar uma função chamada **criarPedidoHistorico()**, onde implementaremos essas funções.

Vamos entender a estrutura que a nossa função "criarPedidoHistorico()" terá:

```
src > JS pedidos.js > ...
1  import { lerLocalStorage, desenharProdutoCarrinhoSimples } from './utilidades';
2
3  function criarPedidoHistorico(pedidoComData) {
4      const elementoPedido = `

${pedidoComData.dataPedido}</p>
5      <section id="container-pedidos-${pedidoComData.dataPedido}"></section>`;
6      const main = document.getElementsByTagName("main")[0];
7      main.innerHTML += elementoPedido;
8
9      for (const idProduto in pedidoComData.pedido) {
10         desenharProdutoCarrinhoSimples(idProduto, `container-pedidos-${pedidoComData.dataPedido}`, pedidoComData.pedido[idProduto]);
11     }
12 }


```

A função **criarPedidoHistorico** é responsável por montar a estrutura de exibição de um pedido no histórico de pedidos na página de pedidos.

Página de Pedidos

Estamos criando uma string chamada **elementoPedido**. Essa string contém HTML que representa o layout de exibição de um pedido no histórico. O **`${pedidoComData.dataPedido}`** é uma interpolação que insere a data do pedido nessa string.

`const main = document.getElementsByTagName("main")[0]; main.innerHTML += elementoPedido;` - Selecionamos o elemento **main** na página e adicionamos o **elementoPedido** à sua propriedade **innerHTML**. Isso insere a estrutura HTML do pedido na página.

Percorrendo os produtos do pedido através do loop **for...in**. Para cada produto, estamos chamando a função **desenharProdutoCarrinhoSimples**, que é usada para renderizar o produto no carrinho, e passamos o ID do produto, o ID do contêiner onde ele deve ser desenhado e a quantidade.

Acabamos de construir a lógica para um cartão de produto, agora precisamos criar uma funcionalidade que renderize todos os cartões na página de pedidos, que será a função **"renderizarHistoricoPedidos()"**.

A função **renderizarHistoricoPedidos** lê o histórico de pedidos armazenado no Local Storage, depois itera por cada pedido com data usando um loop **for...of** e chama a função **criarPedidoHistorico** para exibir cada pedido na página de pedidos. Finalmente, a função é invocada, renderizando o histórico de pedidos na página.

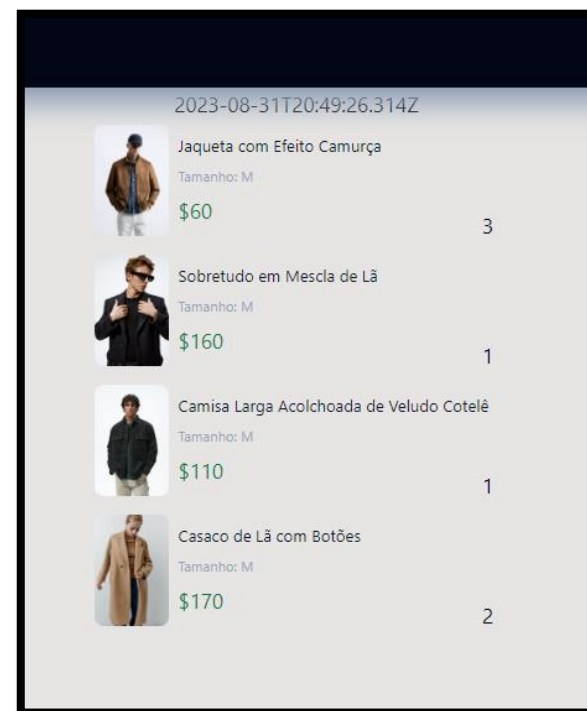
Página de Pedidos

```
function renderizarHistoricoPedidos() {  
  const historico = lerLocalStorage("historico");  
  for (const pedidoComData of historico) {  
    criarPedidoHistorico(pedidoComData);  
  }  
}  
  
renderizarHistoricoPedidos();
```

Para a função ser aplicada no site, precisamos importá-la no arquivo pedidos.html.

```
<main class="flex flex-col items-center"></main>  
<script type="module" src="./src/pedidos.js"></script>  
</body>
```

Com a implementação dessas funcionalidades já conseguimos observar a nossa página de pedidos renderizando os produtos do histórico.



Página de Pedidos

Agora vamos estilizar os produtos na página de pedido com classes do tailwind e formatar a data, para isso vamos usar a expressão **`new Date(pedidoComData.dataPedido).toLocaleDateString()`** faz o seguinte:

- **`new Date(pedidoComData.dataPedido)`**: Isso cria um novo objeto de data usando a informação de data contida em **`pedidoComData.dataPedido`**.
- **`.toLocaleDateString()`**: Isso formata a data em uma string legível para humanos, seguindo o formato de data da região do usuário. Isso significa que a data será exibida de acordo com o idioma e o estilo de data preferidos pelo usuário, tornando-a mais compreensível.

```
src > JS pedidos.js > ...
1  import { lerLocalStorage, desenharProdutoCarrinhosSimples } from './utilidades';
2
3  function criarPedidoHistorico(pedidoComData) {
4    const elementoPedido = `<p class="text-lg text-bold my-4">${new Date(pedidoComData.dataPedido).toLocaleDateString()}</p>
5    <section id="container-pedidos-${pedidoComData.dataPedido}" class="bg-slate-300 p-3 rounded-md"></section>`;
6    const main = document.getElementsByTagName("main")[0];
7    main.innerHTML += elementoPedido;
8
9    for (const idProduto in pedidoComData.pedido) {
10     desenharProdutoCarrinhosSimples(idProduto, `container-pedidos-${pedidoComData.dataPedido}`, pedidoComData.pedido[idProduto]);
11   }
12 }
13
14 function renderizarHistoricoPedidos() {
15   const historico = lerLocalStorage("historico");
16   for (const pedidoComData of historico) {
17     criarPedidoHistorico(pedidoComData);
18   }
19 }
20
21 renderizarHistoricoPedidos();
```

Página de Pedidos

Precisamos fazer mais uma formatação na nossa data e aplicar mais alguns estilos para padronizarmos o lançamentos dos cartões na página de pedidos. A data será formatada para mostrar hora e minuto além de verificar o horário do Brasil.

```
const elementoPedido = `<p class="text-lg text-bold my-4">${new Date(pedidoComData.dataPedido).toLocaleDateString("pt-BR", { hour: "2-digit", minute: "2-digit"})}</p>
```

E vamos aplicar os estilos dentro do arquivo utilidades.js na função `desenharProdutoCarrinhoSimples()`, na variável `articleClasses`.

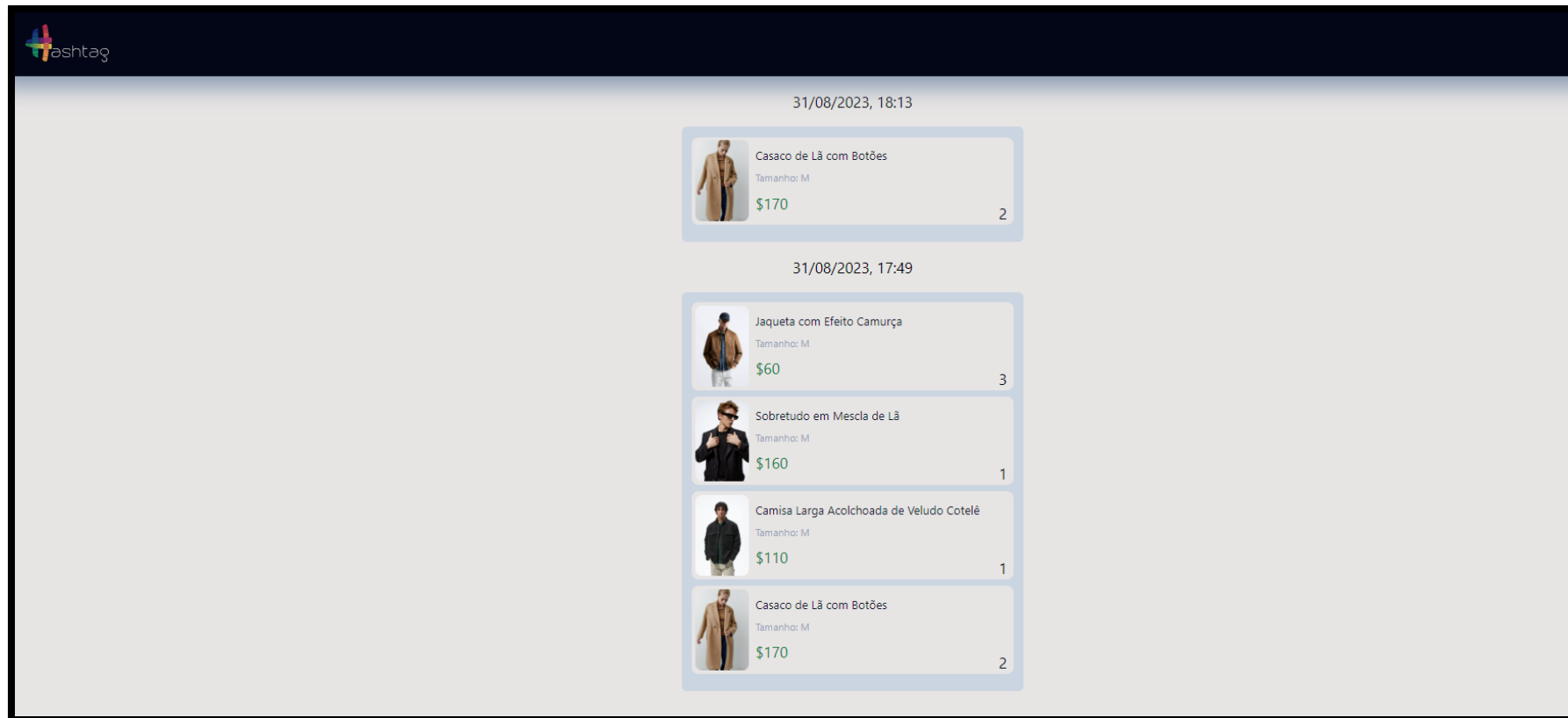
```
const articleClasses = ["flex", "bg-stone-200", "rounded-lg", "p-1", "relative", "mb-2", "w-96"];
```

Além disso, vamos adicionar a tag `<a>` ao redor da tag ``, que contém o logotipo do cabeçalho. Essa tag `<a>` terá o atributo `href="/"`, que fará com que o logotipo funcione como um link para retornar à página principal do site de comércio eletrônico.

```
<a href="/">
  
</a>
```

Página de Pedidos

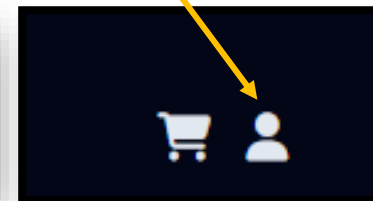
Com isso temos um histórico de pedidos funcional:



Página de Pedidos

Para finalizar, precisamos adicionar o caminho ao contrário, caso eu esteja na página principal e queira consultar o meu histórico de pedidos. Vamos adicionar a **tag <a>** envolta do elemento botão no **arquivo index.html** e adicionar o caminho para a página de pedidos.

```
<a href="/pedidos.html">  
  <button class="mr-2"><i class="fa-solid fa-user"></i></button>  
</a>
```

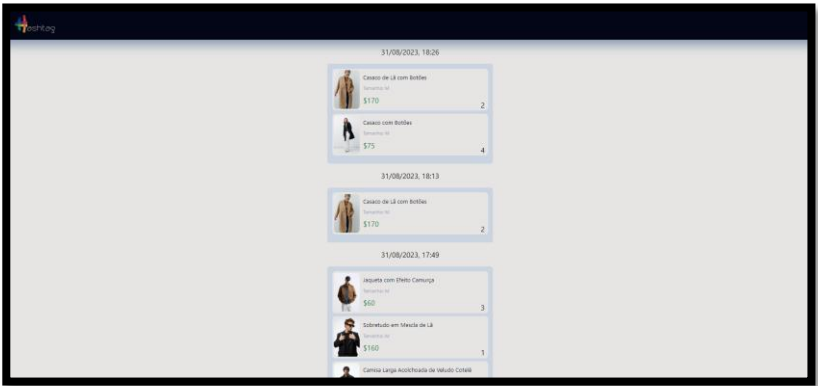
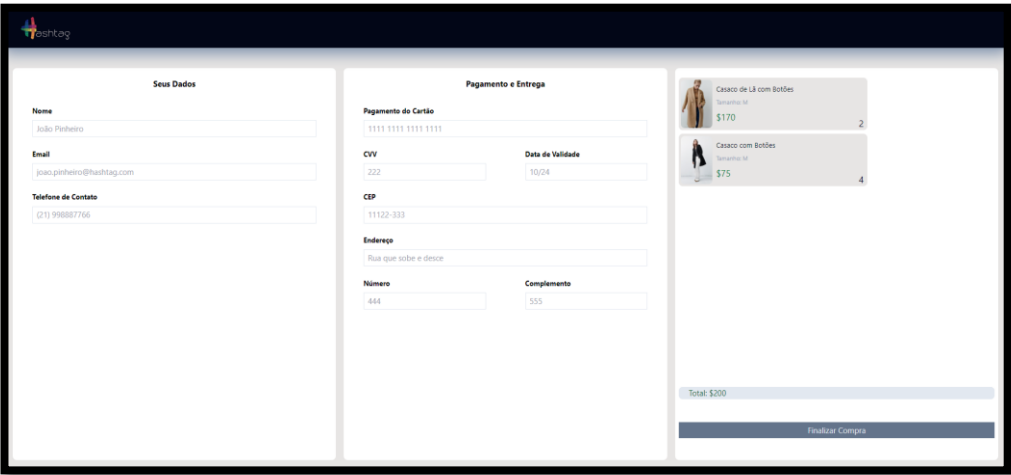
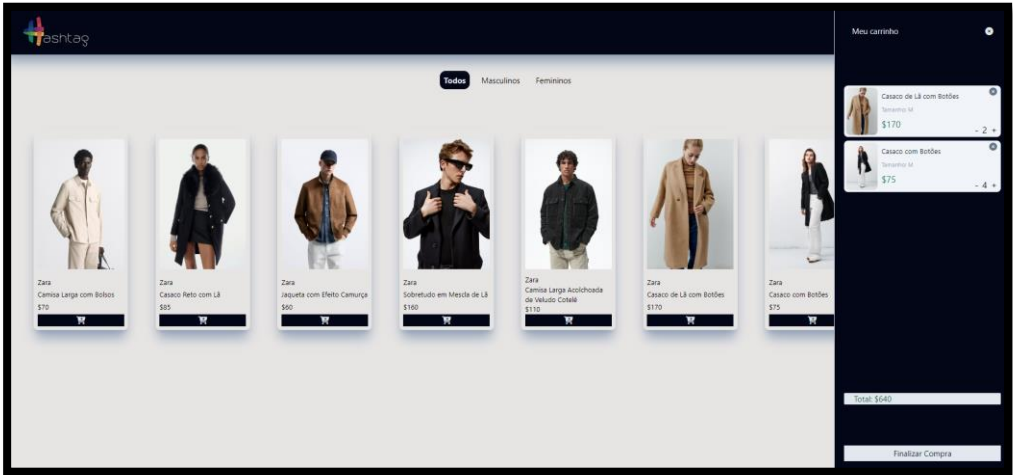


E com isso concluímos o nosso projeto E-Commerce!

Parte 7

Página de Pedidos

Parabéns por concluir o projeto do site de E-commerce! Você demonstrou habilidades valiosas no desenvolvimento web e na criação de uma plataforma de compra online. Este é apenas o começo de sua jornada na programação e design. Continue explorando, aprendendo e aprimorando suas habilidades.



Ainda não segue a gente no Instagram e nem é inscrito no nosso canal do Youtube? Então corre lá!



@hashtagtreinamentos



youtube.com/hashtag-treinamentos

