FONT-ICONS:

<https://flowbite.com/icons/>

CART ICON:

<svg class="w-6 h-6 text-gray-800 dark:text-white" aria-hidden="true" xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 18 20">

<path stroke="currentColor" stroke-linecap="round" stroke-linejoin="round" stroke-width="2" d="M6 15a2 2 0 1 0 0 4 2 2 0 0 0 0-4Zm0 0h8m-8 0-1-4m9 4a2 2 0 1 0 0 4 2 2 0 0 0 0-4Zm-9-4h10l2-7H3m2 7L3 4m0 0-.792-3H1"/>

</svg>

Section: product display

This CSS code is styling a list of products displayed in a grid. Let's break down the styles:

Css-code:

.listProduct {

display: grid;

grid-template-columns: repeat(3, 1fr);

gap: 20px;

}

1. Display Grid (display: grid;):

This property sets the display type of the element with the class .listProduct to be a grid container.

1. Grid Template Columns (grid-template-columns: repeat(3, 1fr);):

This property defines the number and size of columns in the grid. It specifies three columns (repeat(3)) with equal widths (1fr each).

1. Gap (gap: 20px;):

This property sets the gap (spacing) between grid items to be 20 pixels.

Css-code:

.listProduct .item img {

width: 100%;

height: 430px;

object-fit: cover;

}

Image Styles (width, height, object-fit):

* Styles for images within elements with the class .item inside .listProduct.
* Sets the width to 100%, making the image fill its container.
* Sets the height to 430 pixels.
* Uses object-fit: cover; to ensure the image covers the entire container while maintaining its aspect ratio.

Css-code:

.listProduct .item {

position: relative;

}

Item Styles (position: relative;):

Styles for elements with the class .item inside .listProduct.

Setting position: relative; allows absolute positioning of child elements relative to this parent.

Css-code:

.listProduct .item h2 {

font-weight: 700;

font-size: x-large;

}

Header Styles (font-weight, font-size):

Styles for h2 elements within elements with the class .item. The font weight is set to 700 (bold), and the font size is set to x-large.

Css-code:

.listProduct .item .price {

font-size: x-large;

}

Price Styles (font-size):

Styles for elements with the class .price inside elements with the class .item. The font size is set to x-large.

Css-code:

.listProduct .item button {

position: absolute;

top: 50%;

left: 50%;

background-color: #e6572c;

color: #fff;

width: 50%;

border: none;

padding: 20px 30px;

box-shadow: 0 10px 50px #000;

cursor: pointer;

transform: translateX(-50%) translateY(100px);

opacity: 0;

}

Button Styles (position, top, left, background-color, etc.):

* Styles the button within each item.
* Sets the position to absolute, making it positioned relative to the nearest positioned ancestor (in this case, the .item container).
* Positions the button at the center of its parent container using top: 50%; left: 50%; and adjusts its position with transform: translateX(-50%) translateY(100px);.
* Sets background color, text color, width, padding, and box shadow for a stylized appearance.
* Initially sets opacity to 0, making the button invisible.

Css-code:

.listProduct .item:hover button {

transition: 0.5s;

opacity: 1;

transform: translateX(-50%) translateY(0);

}

Button Hover Styles (transition, opacity, transform):

* Styles the button when the parent .item is hovered.
* Applies a smooth transition over 0.5 seconds for the opacity and transform properties, creating a fade-in effect.
* Sets the opacity to 1, making the button visible.
* Adjusts the transform to bring the button back to its original position (translateY(0)).

**Section: CSS code for shopping cart**

This CSS code is defining styles for a shopping cart interface. Let's break down the key components:

.cart:

Sets the overall styling for the cart container.

Applies a fixed position, making it stay in place while the user scrolls.

Specifies a width of 400 pixels, but with a maximum width of 80% of the viewport width (max-width: 80vw).

Sets the height to 100% of the viewport height (height: 100vh).

Sets the background color to #0E0F11 (a dark color).

Positions it at the top of the viewport (top: 0px).

Initially positions it off-screen to the right (right: -100%) with the intention of animating it into view later.

Uses grid layout with three rows and a 20-pixel gap between them.

Defines a transition effect for the right property over a duration of 1 second.

.cart .buttons .checkout:

Styles the checkout button within the cart.

Sets the background color to #E8BC0E and text color to #000.

.cart h2:

Styles the heading within the cart.

Sets the text color to #E8BC0E.

Adds padding, height, and removes any margin.

.cart .listCart .item:

Styles each item within the cart.

Uses grid layout with three columns: one for an image (50px wide), one for the main content (taking up the available space), and one for the quantity (70px wide).

Aligns items in the center vertically.

Adds a margin at the bottom of each item.

.cart .listCart img:

Styles the images within each cart item.

Sets the width to 100%, making the image fill its container.

Sets the height to 70px.

Uses object-fit: cover; to maintain aspect ratio and cover the container.

Applies a border-radius of 10px for rounded corners.

.cart .listCart .item .quantity:

Styles the quantity section within each cart item.

Uses flex layout to justify content to the end (right-align).

.cart .listCart .item .quantity span:

Styles the span elements within the quantity section.

Sets a fixed width of 50px and centers the text.

.cart .listCart:

Styles the list of cart items.

Adds padding and enables overflow auto with a scrollbar that is hidden (::-webkit-scrollbar { width: 0; }).

.cart .buttons:

Styles the buttons section at the bottom of the cart.

Uses grid layout with two columns of equal width.

Aligns text to the center.

.cart .buttons div:

Styles the individual buttons within the buttons section.

Sets the background color to #000, aligns content to the center, sets font weight to bold, and uses a pointer cursor.

.cart .buttons a:

Styles the anchor elements within the buttons section.

Sets text color to #fff and removes text decoration.

In summary, this CSS is designed to create a visually appealing and responsive shopping cart interface, with attention to grid layout, color schemes, and transitions. The cart initially appears off-screen and can slide into view with a smooth animation.

Section: JavaScript code for shopping cart

This JavaScript code is implementing functionality for a simple shopping cart interface. Let's break down what each part does:

Selection of DOM elements:

Javascript-code:

let iconCart = document.querySelector(".iconCart");

let cart = document.querySelector(".cart");

let container = document.querySelector(".container");

let close = document.querySelector(".close");

It selects DOM elements using document.querySelector based on their respective classes. Presumably, iconCart refers to an element that, when clicked, triggers the opening of the shopping cart (cart). container might be the main content container, and close could be a close button within the shopping cart.

Event listener for opening/closing the cart:

Javascript-code:

iconCart.addEventListener("click", function () {

if (cart.style.right == "-100%") {

cart.style.right = "0";

container.style.transform = "translateX(-400px)";

} else {

cart.style.right = "-100%";

container.style.transform = "translateX(0)";

}

});

* It adds a click event listener to the element with the class "iconCart" (presumably, a cart icon).
* The event listener checks the current position of the shopping cart (cart) using the right style property.
* If the cart is currently off-screen (right equals "-100%"), it sets the right property to "0" (bringing the cart into view) and translates the content container (container) to the left by 400 pixels using transform: "translateX(-400px)".
* If the cart is already in view, it hides the cart (sets right to "-100%") and resets the content container's translation (transform: "translateX(0)").

Event listener for closing the cart:

Javascript-code:

close.addEventListener("click", function () {

cart.style.right = "-100%";

container.style.transform = "translateX(0)";

});

It adds a click event listener to the element with the class "close" (presumably, a close button within the cart).

When clicked, it hides the cart **(sets right to "-100%")** and resets the content container's translation **(transform: "translateX(0)")**.

In summary, this JavaScript code creates interactivity for opening and closing a shopping cart. When the cart icon is clicked, it toggles the visibility of the cart and translates the content container accordingly. The close button within the cart is also wired to close the cart and reset the content container's position.

**Section: JS code fetches data from a JSON file and to process the data**

This JavaScript code fetches data from a JSON file (product.json), processes the data, and then dynamically adds product information to an HTML element with the class "listProduct." Here's a step-by-step explanation:

Initialization:

JS- code

let products = null;

* Initializes a variable products and sets it to null. This variable will later be used to store the product data retrieved from the JSON file.

Fetching data from a JSON file:

JS- code

fetch("product.json")

.then((response) => response.json())

.then((data) => {

products = data;

addDataToHTML();

});

* Uses the fetch API to make an asynchronous request to the "product.json" file.
* The then method is used to handle the response. It first converts the response to JSON using response.json().
* The second then block sets the products variable to the retrieved data and calls the addDataToHTML function.

Function to add data to HTML:

JS- code

function addDataToHTML() {

let listProductHTML = document.querySelector(".listProduct");

listProductHTML.innerHTML = "";

if (products != null) {

products.forEach((product) => {

let newProduct = document.createElement("div");

newProduct.classList.add("item");

newProduct.innerHTML = `<img src="${product.image}" alt="">

<h2>${product.name}</h2>

<div class="price">$${product.price}</div>

<button onclick="addCart(${product.id})">Add To Cart</button>`;

listProductHTML.appendChild(newProduct);

});

}

}

* Selects the HTML element with the class "listProduct."
* Clears any existing content inside this element (listProductHTML.innerHTML = "";).
* Checks if there is data in the products variable.
* If there is data, it iterates through each product in the array (products.forEach).
* Creates a new div element for each product, adds the necessary HTML content (image, name, price, and a button with an onclick event calling addCart function), and appends this new element to the "listProduct" container.

In summary, this script fetches product data from a JSON file, processes it, and dynamically populates an HTML element with class "listProduct" with product information. The addDataToHTML function is responsible for updating the HTML content based on the retrieved product data.

**Section: JS code for shopping cart functionality using cookies to persist the cart data**

This JavaScript code implements a simple shopping cart functionality using cookies to persist the cart data across page refreshes. Let's break down each part:

Cookie Handling Functions:

JS- code

let listCart = [];

function checkCart() {

var cookieValue = document.cookie

.split('; ')

.find(row => row.startsWith('listCart='));

if (cookieValue) {

listCart = JSON.parse(cookieValue.split('=')[1]);

} else {

listCart = [];

}

}

checkCart();

* Defines an empty array listCart to store the cart items.
* checkCart function reads the cookie named 'listCart' and parses its value. If the cookie exists, it populates listCart with the parsed data; otherwise, it initializes listCart as an empty array.

Add to Cart Function:

JS- code

function addCart($idProduct) {

let productsCopy = JSON.parse(JSON.stringify(products));

if (!listCart[$idProduct]) {

listCart[$idProduct] = productsCopy.filter(product => product.id == $idProduct)[0];

listCart[$idProduct].quantity = 1;

} else {

listCart[$idProduct].quantity++;

}

document.cookie = "listCart=" + JSON.stringify(listCart) + "; expires=Thu, 31 Dec 2025 23:59:59 UTC; path=/;";

addCartToHTML();

}

addCartToHTML();

* The addCart function takes a product ID ($idProduct) as an argument.
* It creates a deep copy of the products array to avoid modifying the original.
* If the product is not already in the cart, it adds it, sets the quantity to 1. If it is already in the cart, it increments the quantity.
* The function then updates the 'listCart' cookie with the modified data and calls addCartToHTML to update the HTML representation of the cart.

HTML Update Function (Displaying Cart Contents):

JS- code

function addCartToHTML() {

let listCartHTML = document.querySelector('.listCart');

listCartHTML.innerHTML = '';

let totalHTML = document.querySelector('.totalQuantity');

let totalQuantity = 0;

if (listCart) {

listCart.forEach(product => {

if (product) {

// ... (creates and appends HTML elements for each product in the cart)

totalQuantity = totalQuantity + product.quantity;

}

})

}

totalHTML.innerText = totalQuantity;

}

* addCartToHTML function updates the HTML representation of the cart.
* It clears the existing content inside the cart element, iterates through each product in listCart, creates HTML elements for each product, appends them to the cart element, and updates the total quantity display.

Change Quantity of Cart Items Function:

JS- code

function changeQuantity($idProduct, $type) {

switch ($type) {

case '+':

listCart[$idProduct].quantity++;

break;

case '-':

listCart[$idProduct].quantity--;

if (listCart[$idProduct].quantity <= 0) {

delete listCart[$idProduct];

}

break;

default:

break;

}

document.cookie = "listCart=" + JSON.stringify(listCart) + "; expires=Thu, 31 Dec 2025 23:59:59 UTC; path=/;";

addCartToHTML();

}

* The changeQuantity function is called when the user clicks the "+" or "-" buttons to change the quantity of an item in the cart.
* It takes the product ID and the type of change ("+" or "-") as arguments.
* It updates the quantity of the specified product in the listCart array.
* If the quantity becomes zero or less, it removes the product from the cart.
* It then updates the "listCart" cookie and refreshes the HTML representation of the cart.

In summary, this code provides a basic shopping cart functionality where cart data is stored and retrieved from cookies, allowing the cart contents to persist across page reloads. The script also updates the HTML to reflect changes in the cart.

Section: Html-code for checkout page for a shopping cart

This HTML document represents a simple checkout page for a shopping cart. Here's a breakdown of its structure and content:

Html- code

<div class="container">

<div class="checkoutLayout">

<!-- Left Section: Product List in Cart -->

<div class="returnCart">

<!-- "Keep Shopping" link and heading -->

<a href="/">keep shopping</a>

<h1>List Product in Cart</h1>

<!-- Product list with image, name, price, quantity, and total price -->

<div class="list">

<!-- Sample item -->

<div class="item">

<img src="images/1.webp" />

<div class="info">

<div class="name">PRODUCT 1</div>

<div class="price">$22/1 product</div>

</div>

<div class="quantity">5</div>

<div class="returnPrice">$433.3</div>

</div>

<!-- More items can be added dynamically -->

</div>

</div>

<!-- Right Section: Checkout Form -->

<div class="right">

<h1>Checkout</h1>

<!-- Form with input fields for name, phone, address, country, and city -->

<div class="form">

<!-- Form fields go here -->

</div>

<!-- Return section showing total quantity and total price -->

<div class="return">

<div class="row">

<div>Total Quantity</div>

<div class="totalQuantity">70</div>

</div>

<div class="row">

<div>Total Price</div>

<div class="totalPrice">$900</div>

</div>

</div>

<!-- Checkout button -->

<button class="buttonCheckout">CHECKOUT</button>

</div>

</div>

</div>

Contains two main sections within a container:

**Left Section (returnCart):** Displays the list of products in the cart.

**Right Section (right):** Contains a checkout form with input fields for name, phone, address, country, and city, as well as a return section showing the total quantity and total price. It also includes a "CHECKOUT" button.

Note: The product list is currently hardcoded, but it can be dynamically generated based on the actual contents of the shopping cart.

Script Link:

Html- code

<script src="checkout.js"></script>

Links an external JavaScript file (checkout.js) to provide dynamic functionality and interaction with the page.

In summary, this HTML document represents a basic checkout page with sections for displaying products in the cart and a form for collecting user information before checkout. The layout is structured, and additional styling and functionality are expected to be provided by the linked stylesheet (style.css) and JavaScript file (checkout.js).

**Section: js code checkout function(checkout.js)**

This code appears to be part of a web application that manages a shopping cart. Let's break down the code step by step:

Initialize listCart and Check for Existing Cart Data:

JS-code

let listCart = [];

function checkCart() {

var cookieValue = document.cookie

.split("; ")

.find((row) => row.startsWith("listCart="));

if (cookieValue) {

listCart = JSON.parse(cookieValue.split("=")[1]);

}

}

checkCart();

* listCart is initialized as an empty array.
* The checkCart function reads the value of a cookie named "listCart" and updates listCart if the cookie exists. The cookie likely stores the shopping cart data in JSON format.

Update HTML Representation of the Cart:

JS-code

addCartToHTML();

Immediately after checking the cart, the addCartToHTML function is called to update the HTML representation of the shopping cart.

addCartToHTML Function:

JS-code

function addCartToHTML() {

// clear data default

let listCartHTML = document.querySelector(".returnCart .list");

listCartHTML.innerHTML = "";

let totalQuantityHTML = document.querySelector(".totalQuantity");

let totalPriceHTML = document.querySelector(".totalPrice");

let totalQuantity = 0;

let totalPrice = 0;

// if product is in Cart

if (listCart) {

listCart.forEach((product) => {

if (product) {

let newCart = document.createElement("div");

newCart.classList.add("item");

newCart.innerHTML = `<img src="${product.image}">

<div class="info">

<div class="name">${product.name}</div>

<div class="price">$${product.price}/1 product</div>

</div>

<div class="quantity">${product.quantity}</div>

<div class="returnPrice">$${

product.price \* product.quantity

}</div>`;

listCartHTML.appendChild(newCart);

totalQuantity = totalQuantity + product.quantity;

totalPrice = totalPrice + product.price \* product.quantity;

}

});

}

totalQuantityHTML.innerText = totalQuantity;

totalPriceHTML.innerText = "$" + totalPrice;

}

* The addCartToHTML function updates the HTML representation of the shopping cart based on the contents of listCart.
* It first clears the default data in the cart by emptying the listCartHTML container.
* It then iterates over each product in listCart, creating HTML elements for each and appending them to the cart container.
* The total quantity and total price of the items in the cart are calculated during this process.
* Finally, it updates the total quantity and total price displayed on the webpage.

In summary, this code checks for the existence of a shopping cart stored in a cookie, updates the listCart variable, and then updates the HTML representation of the shopping cart on the webpage, including item details, quantities, and

**Section: CSS code defines styles for a checkout page**

This CSS code defines styles for a checkout page layout. Let's break down the code section by section:

Checkout Layout Styling:

Css-code

.checkoutLayout {

display: grid;

grid-template-columns: repeat(2, 1fr);

gap: 50px;

padding: 20px;

}

The .checkoutLayout class is applied to the container for the checkout layout.

It uses CSS Grid to create a two-column layout with a 50px gap between columns.

The padding is added to the container.

Css-code

.checkoutLayout .right {

background-color: #5358b3;

border-radius: 20px;

padding: 40px;

color: #fff;

}

Styles for the right side of the checkout layout.

It sets the background color, border radius, padding, and text color.

Css-code

.checkoutLayout .right .form {

display: grid;

grid-template-columns: repeat(2, 1fr);

gap: 20px;

border-bottom: 1px solid #7a7fe2;

padding-bottom: 20px;

}

Styles for the form within the right side of the checkout layout.

Again, it uses CSS Grid for a two-column layout with a 20px gap.

Adds a border-bottom and padding to the form.

Css-code

.checkoutLayout .form h1,

.checkoutLayout .form .group:nth-child(-n + 3) {

grid-column-start: 1;

grid-column-end: 3;

}

Styles for the heading and the first three groups in the form.

The grid-column-start and grid-column-end properties make them span across both columns.

Css-code

.checkoutLayout .form input,

.checkoutLayout .form select {

width: 100%;

padding: 10px 20px;

box-sizing: border-box;

border-radius: 20px;

margin-top: 10px;

border: none;

background-color: #6a6fc9;

color: #fff;

}

Styles for input fields and select elements within the form.

It sets their width to 100%, adjusts padding, adds border-radius, removes border, and sets background and text color.

Css-code

.checkoutLayout .right .return .row {

display: flex;

justify-content: space-between;

align-items: center;

margin-top: 10px;

}

.checkoutLayout .right .return .row div:nth-child(2) {

font-weight: bold;

font-size: x-large;

}

Styles for a row within the right side of the checkout layout.

It uses flexbox for layout, justifies content, aligns items, and sets styling for the second child element.

Css-code

.buttonCheckout {

width: 100%;

height: 40px;

border: none;

border-radius: 20px;

background-color: #49d8b9;

margin-top: 20px;

font-weight: bold;

color: #fff;

}

Styles for a checkout button.

It sets width, height, border-radius, background color, margin, and text styling.

Css-code

.returnCart h1 {

border-top: 1px solid #eee;

padding: 20px 0;

}

Styles for a heading within the return cart section.

It adds a border at the top and padding.

Css-code

.returnCart .list .item img {

height: 80px;

}

Styles for images within items in the return cart list.

Sets a fixed height for the images.

Css-code

.returnCart .list .item {

display: grid;

grid-template-columns: 80px 1fr 50px 80px;

align-items: center;

gap: 20px;

margin-bottom: 30px;

padding: 0 10px;

box-shadow: 0 10px 20px #5555;

border-radius: 20px;

}

Styles for items in the return cart list.

Utilizes CSS Grid for a four-column layout, sets alignment, gap, margin, padding, box-shadow, and border-radius.

Css-code

.returnCart .list .item .name,

.returnCart .list .item .returnPrice {

font-size: large;

font-weight: bold;

}

Styles for the name and return price within items.

Sets font size and weight.

In summary, this CSS code defines styles for a responsive checkout page layout, including form elements, buttons, and the appearance of items in the return cart list. The use of CSS Grid and flexbox contributes to creating a visually appealing and organized layout. The styling is done with attention to details such as border radius, padding, colors, and font styles.