LEANNE'S COLLECTION FULL STACK APP

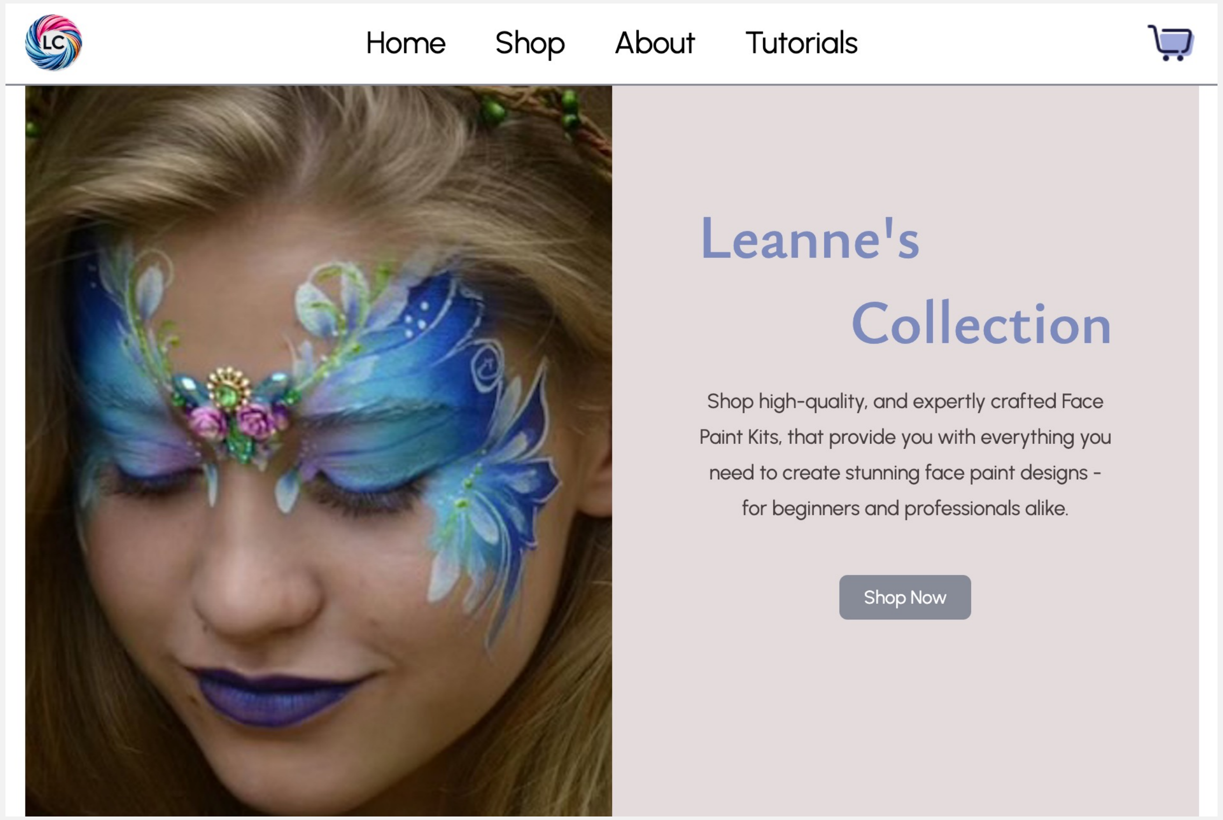
By Laylah De Paull

GCAS022002

Presentation Link

PROJECT OVERVIEW - CUSTOMER

Link To Video Overview

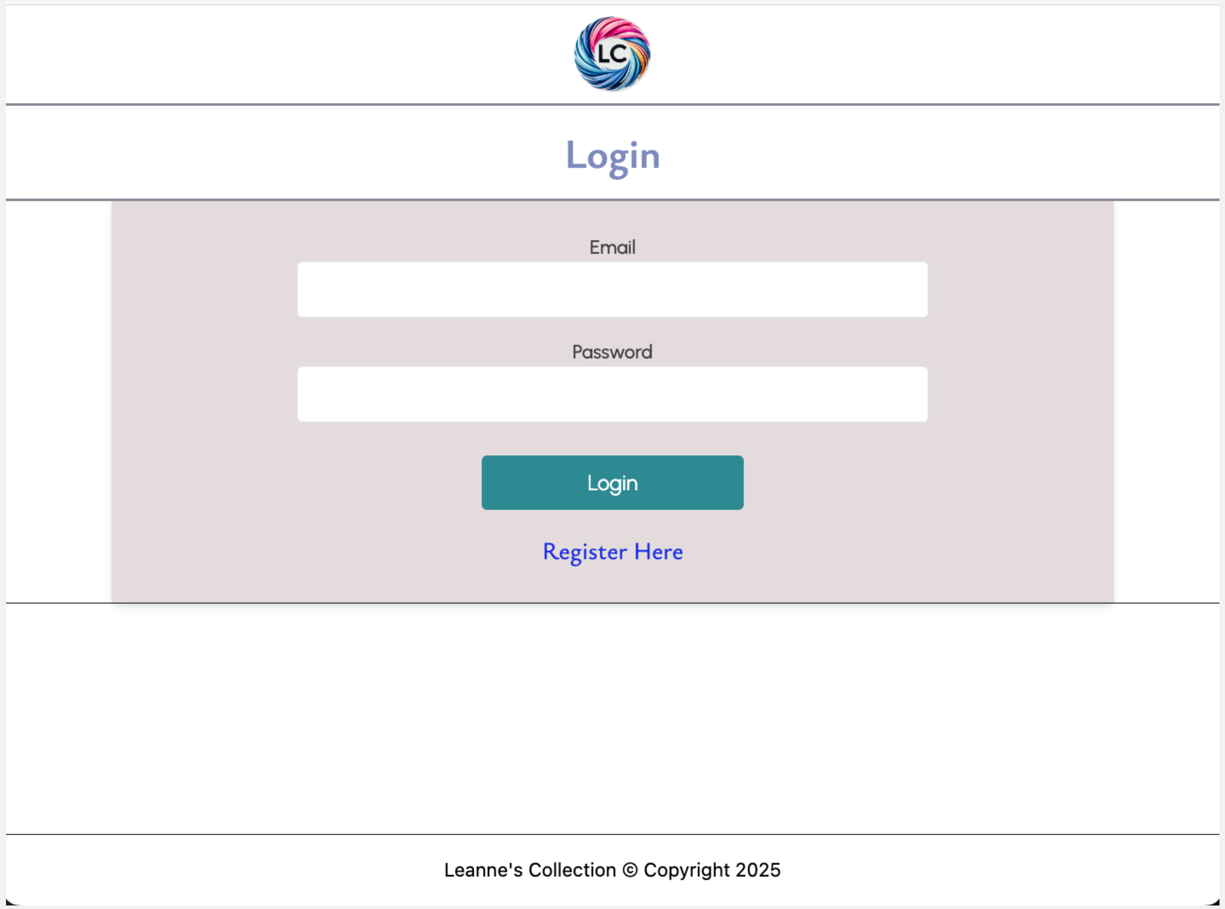


Shop high-quality, and expertly crafted Face Paint Kits, that provide you with everything you need to create stunning face paint designs - for beginners and professionals alike.

- The MERN stack application is called "Leanne's Collection". It is a responsive eCommerce website built with MongoDB, Express.js, React, Node.js, and Tailwind CSS, designed to sell Face Paint Kits by Face & Body Paint artist Leanne Courtney
- The Home Page features a striking Hero Section that introduces users to the brand. It showcases a few of the available kits, offers a little information about the artist, and includes a few tutorial videos.
- The Home page also includes call-to-action buttons and navigation links that guide users to the key sections of the site. A footer provides essential shipping and contact details.
- The Shop Page allows users browse all the face paint kits on offer, view more detailed information on a products page, and add kits to their shopping cart. The cart icon also updates in real-time, and Toastify provides notifications when kits are first added or completely removed from the shopping cart.
- The About Page shares background information on Leanne Courtney, with links to contact information like email, phone & and social media. It also includes information on newly released products.
- The Tutorials Page lets users browse and watch short videos that explain how to use the face paint kits that are being sold, offering handy hits and guidance about how to use the kits.
- The site also includes, Cart, Checkout, & Order Success pages. On the Cart Page, users can increase or decrease kit quantities or remove kits completely. On the Checkout Page, users enter their shipping details and can submit an order. After submitting, users are automatically redirected to the Order Success Page to view their order summary.
- Note: Checkout and Order Success pages are currently proxies, and no payment details can be entered. In future development, Shopify APIs will be integrated to handle real cart creation, checkout, and the payment process.

PROJECT OVERVIEW - ADMIN

Link To Video Overview

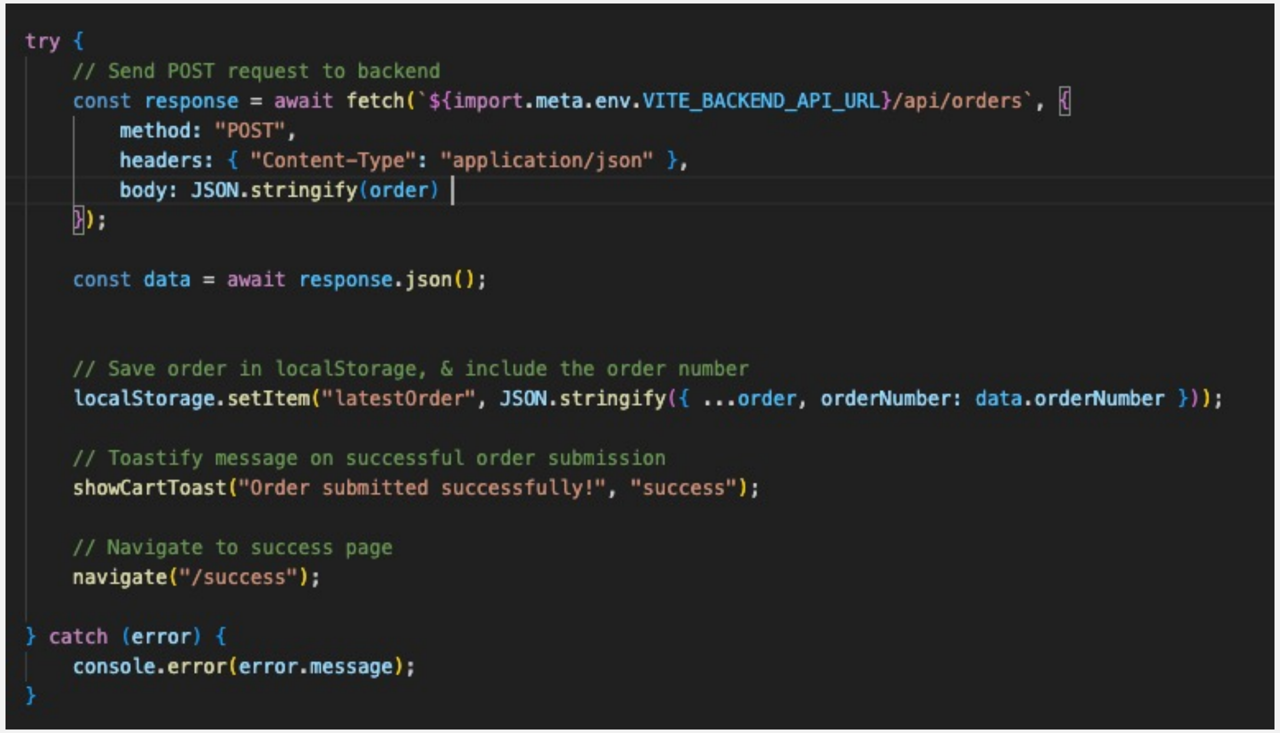


Leanne's Collection © Copyright 2025

- The seller has access to their own admin portal, where they can register an account. Registration however has been limited to only the first two accounts registered.
- The seller can also log in through the that same portal, the form can be toggled between Register & Login. Upon login, a secure JWT token is generated, and has been set to be only valid for one hour.
- After a successful login and database check, the admin is automatically redirected to the orders dashboard, which displays all customer orders made.
- The order dashboard shows essential order information, including the order number, customer's name, total, date placed, and current status. The admin can toggle an orders status between unfulfilled and fulfilled.
- Admins can also click on an orders number, which is a link into that specific order. There the admin can view more detail for that order, such as the customer's email, shipping address, and the itemised order summary.
- A logout button is also available on the order dashboard page, allowing the admin to securely logout and end their session.
- It is important to note that in future development I will also add functionality to allow admins to upload new face paint kits and video tutorials through the admin portal.

NETWORK REQUEST – KEY CODE EXPLAINED

CHECKOUT PAGE

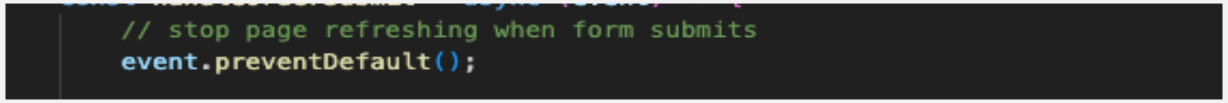


```
try {
    // Send POST request to backend
    const response = await fetch(`${import.meta.env.VITE_BACKEND_API_URL}/api/orders`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(order)
    });

    const data = await response.json();


    // Save order in localStorage, & include the order number
    localStorage.setItem("latestOrder", JSON.stringify({ ...order, orderNumber: data.orderNumber }));

    // Toastify message on successful order submission
    showCartToast("Order submitted successfully!", "success");

    // Navigate to success page
    navigate("/success");

} catch (error) {
    console.error(error.message);
}
```

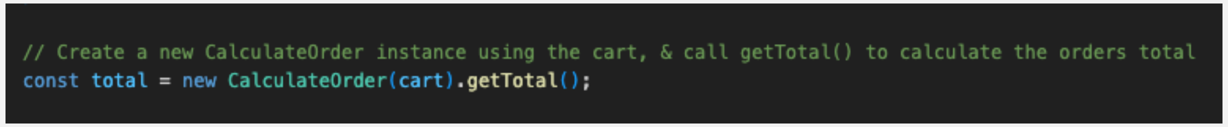
- On the Checkout Page, a customer can submit an order, which is stored in the orders collection of the cloud-based MongoDB Atlas database created for Leanne's Collection.
- A customer's order is submitted through the handleOrderSubmit function, which sends a POST request to the backend API.
- Network requests for order creation are made using fetch(), which targets the backend route /api/orders, defined in the order routes file.
- The request includes a HTTP method, headers, and a body. The method used is POST, the headers specify that the content is a JSON string, and the body contains a stringified version of the order object, that includes the customer's name, email, address, cart items, and total.
- The line const data = await response.json() allows time for the backend to respond, and parses the returned JSON-formatted string into a usable JavaScript object.
- On the backend, the logic for creating the order is handled by the createOrder controller, which uses the Order model.
- The order is stored in the database and a unique order number (starting from 2000) is generated. This number is returned back to the frontend.
- The complete order data, including the generated order number, is then saved to localStorage and used to generate the order summary shown to the customer when they are re-directed after a successful checkout to the the order success page.
- It's also important to note that the order number is also especially important for the Admin. That is; it helps establish a clear order chronology, help differentiate between orders, and provides a link to each specific order for more detailed information.

FUNCTION CALLS – KEY CODE EXPLAINED

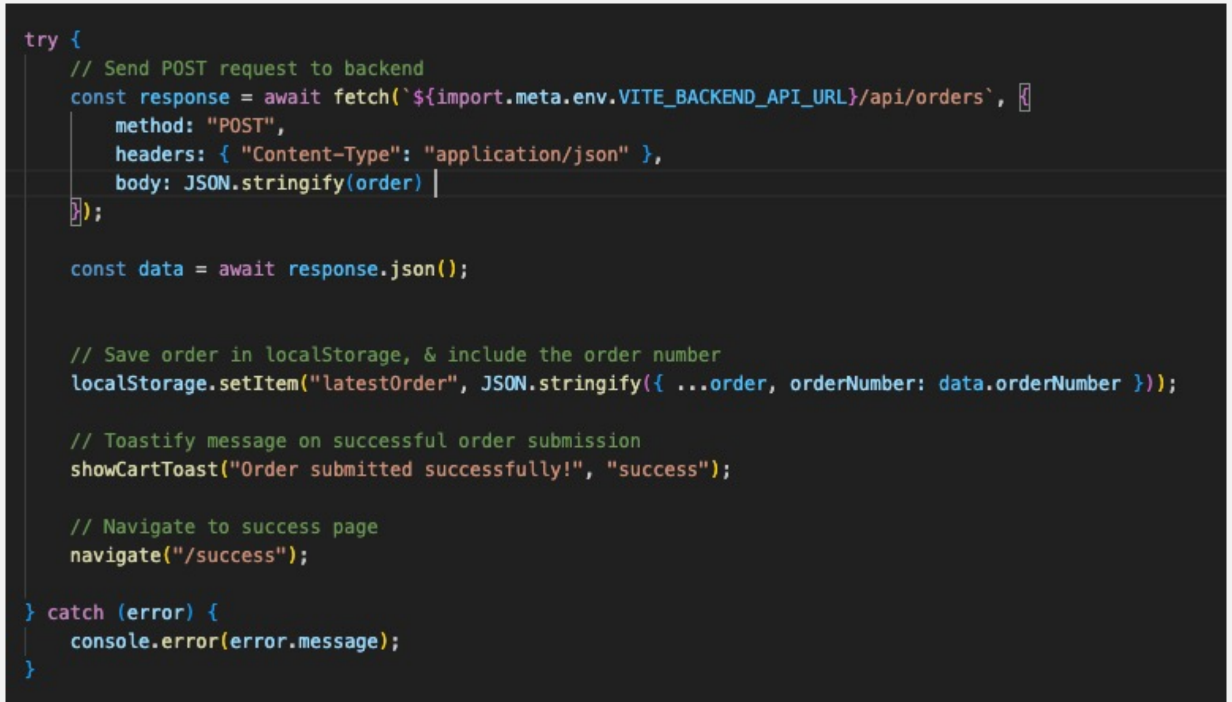
CHECKOUT PAGE



```
    // stop page refreshing when form submits
    event.preventDefault();
```



```
// Create a new CalculateOrder instance using the cart, & call getTotal() to calculate the orders total
const total = new CalculateOrder(cart).getTotal();
```

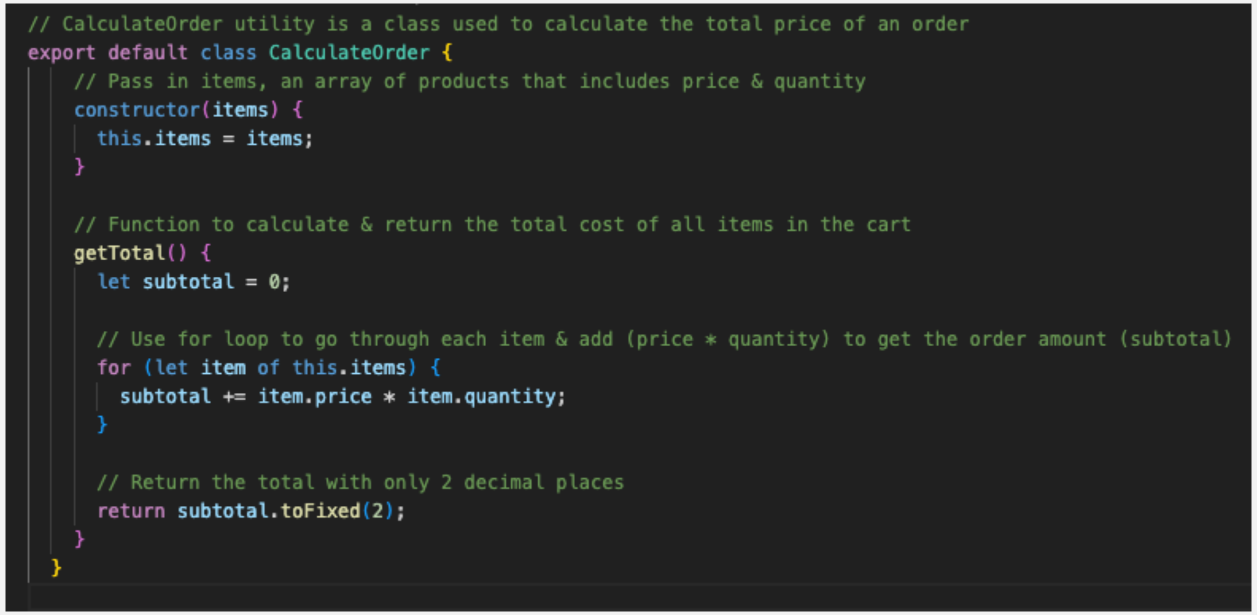


```
try {
    // Send POST request to backend
    const response = await fetch(`${import.meta.env.VITE_BACKEND_API_URL}/api/orders`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(order)
    });

    const data = await response.json();


    // Save order in localStorage, & include the order number
    localStorage.setItem("latestOrder", JSON.stringify({ ...order, orderNumber: data.orderNumber }));

    // Toastify message on successful order submission
    showCartToast("Order submitted successfully!", "success");

    // Navigate to success page
    navigate("/success");

} catch (error) {
    console.error(error.message);
}
```

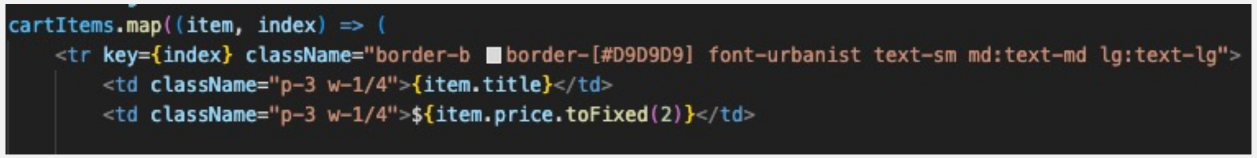
- On the Checkout Page, within the handleOrderSubmit function, there are four key elements being used: event.preventDefault(), a custom utility class CalculateOrder, a custom function showCartToast(), and the React Router function navigate("/success").
- The event.preventDefault() method/function is used on the Checkout Page to stop the browser's default form submission behaviour, which would otherwise refresh the page. This ensures the customer's shipping data can be validated, the order request to be sent, and the customer redirected to the order success page, all without reloading occurring.
- The CalculateOrder class is a custom utility for calculating the cart's total. It includes a .getTotal() method, which is a type of function that uses a loop to multiply each item's price by its quantity, adds them together, and returns the final total.
- The CalculateOrder class utility is also reused in other components like CartTable, ensuring consistent total calculations across the app.
- The second element is showCartToast(), which is another custom utility function that triggers a styled toast notification banner. In the checkout page context, it displays a teal toast notification to the customer to let them know their order was successfully submitted.
- The showCartToast() function displays custom toast notifications styled with different colours depending on the type of action. It uses an if / else if / else conditional to check the type argument passed in, such as "remove" for red, or "success" for teal. This logic allows the same function to be reused throughout the app for different events, like removing items, submitting an order, or logging in/out as admin.
- The third function call is navigate("/success"), which uses React Router's useNavigate() hook. Which is used to automatically redirect a customer to the Order Success page once their order has been submitted.
- Using a method like preventDefault(), along with a class, utility functions, and a navigation function helps keep my application logic DRY, reusable, and easier to maintain. This supports a clean and modular codebase that's easier to update and understand.

LOOP – KEY CODE EXPLAINED

CHECKOUT PAGE



```
// CalculateOrder utility is a class used to calculate the total price of an order
export default class CalculateOrder {
    // Pass in items, an array of products that includes price & quantity
    constructor(items) {
      this.items = items;
    }

    // Function to calculate & return the total cost of all items in the cart
    getTotal() {
      let subtotal = 0;

      // Use for loop to go through each item & add (price * quantity) to get the order amount (subtotal)
      for (let item of this.items) {
        subtotal += item.price * item.quantity;
      }

      // Return the total with only 2 decimal places
      return subtotal.toFixed(2);
    }
  }
```

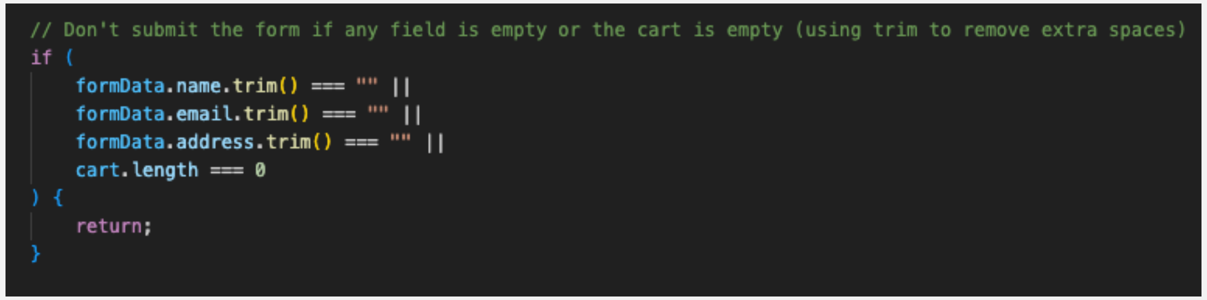


```
cartItems.map((item, index) => (
    <tr key={index} className="border-b border-[#D9D9D9] font-urbanist text-sm md:text-md lg:text-lg">
        <td className="p-3 w-1/4">{item.title}</td>
        <td className="p-3 w-1/4">${item.price.toFixed(2)}</td>
```

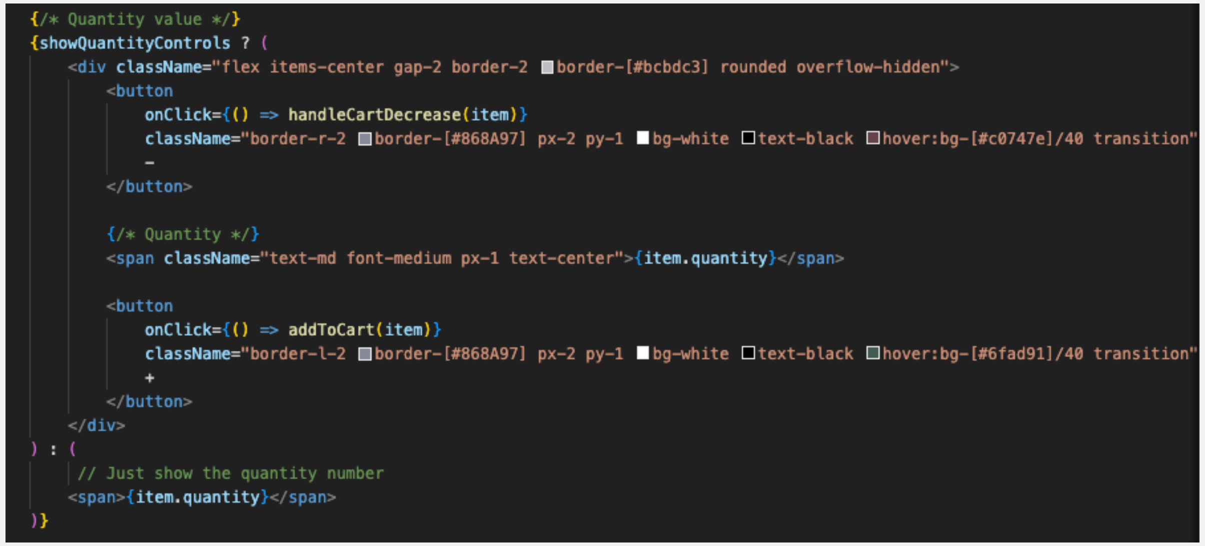
- On the Checkout Page, within the handleOrderSubmit function, a custom utility class called CalculateOrder is used, where the .getTotal() method is called with the cart as an argument.
- The CalculateOrder class is one I created to calculate the total cost of all items in a customer's shopping cart. It is also reused in the CartTable component.
- The .getTotal() method inside this class uses a for...of loop to iterate through each item in the cart (this.items) and multiplies each item's price by its quantity.
- The resulting values are added to a running subtotal, which is then returned with two decimal places using the .toFixed(2) method.
- Because this logic is inside a class, the this keyword is used to access the items passed into the constructor.
- This loop ensures that the total cost is accurately calculated every time it is needed, whether it's on the Cart Page for a subtotal display or on the Checkout Page when submitting the final order.
- The Checkout Page also includes a component called CartTable, which displays all cart items in a clear, table-like layout.
- The CartTable component uses the .map() method to loop over all items in the shopping cart and dynamically render each item's name, price, quantity, and total.
- This loop is important as it allows customers to review their shopping cart before submitting their order.

CONDITIONAL STATEMENT – KEY CODE EXPLAINED

CHECKOUT PAGE



```
// Don't submit the form if any field is empty or the cart is empty (using trim to remove extra spaces)
if (
    formData.name.trim() === "" ||
    formData.email.trim() === "" ||
    formData.address.trim() === "" ||
    cart.length === 0
) {
    return;
}
```

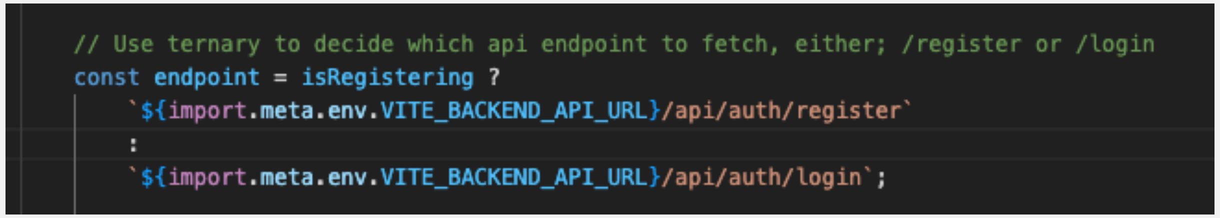


```
{/* Quantity value */}
{showQuantityControls ? (
    <div className="flex items-center gap-2 border-2 border-[#bcbdc3] rounded overflow-hidden">
        <button
            onClick={() => handleCartDecrease(item)}
            className="border-r-2 border-[#868A97] px-2 py-1 bg-white text-black hover:bg-[#c0747e]/40 transition"
            -
        </button>

        {/* Quantity */}
        <span className="text-md font-medium px-1 text-center">{item.quantity}</span>

        <button
            onClick={() => addToCart(item)}
            className="border-l-2 border-[#868A97] px-2 py-1 bg-white text-black hover:bg-[#6fad91]/40 transition"
            +
        </button>
    </div>
) : (
    // Just show the quantity number
    <span>{item.quantity}</span>
)}
```

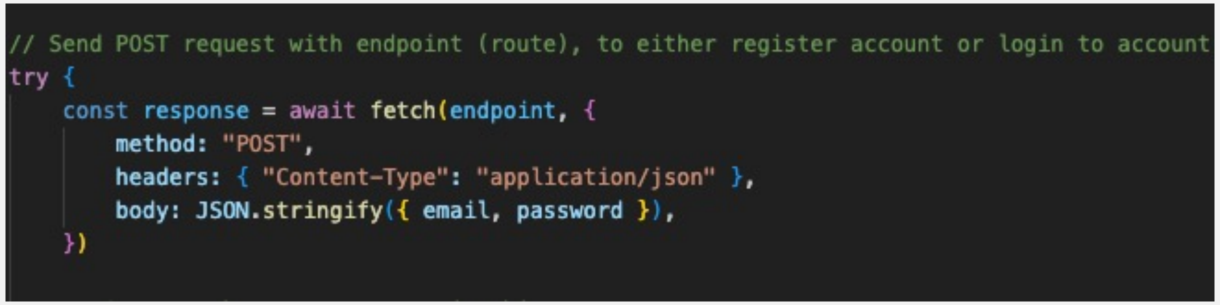
- On the Checkout Page, a customer is required to complete a form with their shipping details, including their name, email, and address.
- Within the handleOrderSubmit function, there is an important conditional statement that prevents the order from being submitted if any of the shipping fields are empty or if the cart is empty.
- This conditional uses an if statement to check whether any input field is empty or the cart length is zero. If any condition is true, the function uses return to exit early, preventing the submission.
- The .trim() method is applied to each form field to ensure that customers do not accidentally submit blank spaces instead of valid shipping information.
- This conditional is essential for preventing the frontend from submitting invalid orders missing critical customer data like name, email, or address. It stops the request early and avoids unnecessary network traffic. However, the backend also includes its own validation to ensure no incomplete orders are accepted, even if this frontend check was bypassed.
- The Checkout Page also uses a reusable component called CartTable, which contains another conditional statement using a ternary operator: showQuantityControls ?.
- The ternary checks if showQuantityControls is true (default). If it is, the (+/-) quantity control buttons are shown. If it is false, only the item quantity is displayed with no quantity control buttons.
- On the Checkout Page, quantity controls are hidden intentionally to finalise the order, unlike on the Cart Page where the customer can still adjust the quantities of the items in their cart before proceeding to checkout.
- This logic ensures that the CartTable behaves differently based on the context it's used in, helping keep the component reusable across my app and my overall codebase DRY and maintainable.

NETWORK REQUEST – PROBLEM CODE EXPLAINED

LOGIN PAGE



```
// Use ternary to decide which api endpoint to fetch, either; /register or /login
const endpoint = isRegistering ?
    `${import.meta.env.VITE_BACKEND_API_URL}/api/auth/register`
    :
    `${import.meta.env.VITE_BACKEND_API_URL}/api/auth/login`;
```

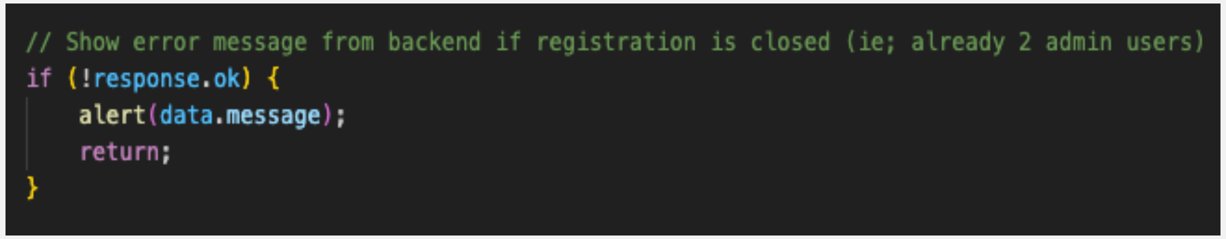


```
// Send POST request with endpoint (route), to either register account or login to account
try {
    const response = await fetch(endpoint, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ email, password }),
    })
```

- This fetch request is used for both login and register actions, depending on the isRegistering state, which selects the correct endpoint (/login or /register) using a ternary.
- My focus here is on the fetch request when a user attempts to register an admin account, which sends the email and password to the backend /api/auth/register route.
- I had written backend logic to only allow two admins to register, with a third request meant to trigger a backend error message and stop further registrations.
- However, during testing, it appeared that more than two users could still register successfully, because a success message appeared, but those users couldn't log in and weren't added to the database.
- The issue was that the frontend wasn't checking the backends response status, so even if the backend rejected the request, the frontend assumed it had succeeded.
- This gave users false feedback, making it appear as though the registration had worked, even though it had actually failed.
- I fixed this by adding a conditional check for !response.ok inside the handleSubmit function, which prevents the success logic from running and instead alerts the user with the correct backend error message.
- This ensured users received accurate feedback when registration was closed, avoiding confusion and keeping the frontend in line with the backend logic.

CONDITIONAL STATEMENT – PROBLEM CODE EXPLAINED

LOGIN PAGE

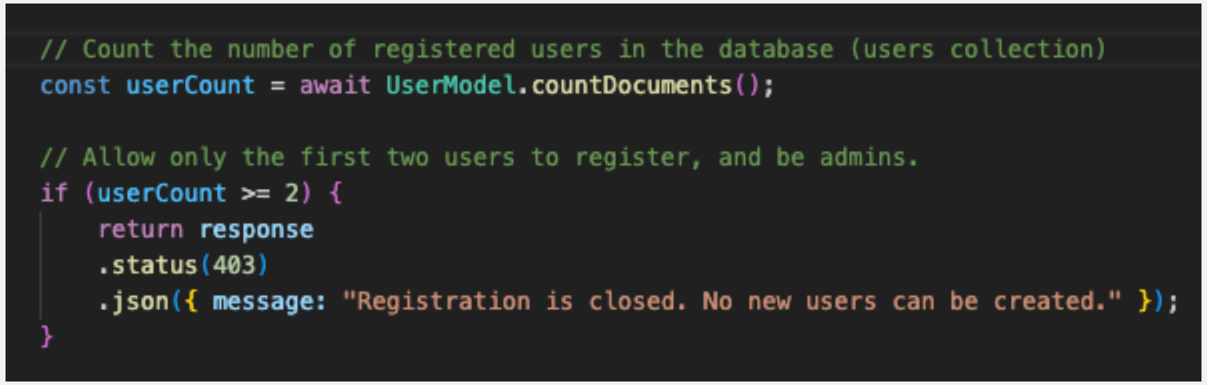


```
// Show error message from backend if registration is closed (ie; already 2 admin users)
if (!response.ok) {
    alert(data.message);
    return;
}
```

- This conditional was added inside the handleSubmit function on the frontend LoginPage to check whether the backend registration request had succeeded or failed.
- Before this condition was added, users were being shown a success message after submitting their admin registration form, even if the backend had rejected the registration due to the 2 admin limit logic I had set up.
- The !response.ok condition checks whether the HTTP status returned by the backend is not in the successful range, that is between: 200–299.
- If the request failed, for example, when trying to create a third admin, this condition triggers the alert() function to display the backend's error message.
- The return statement immediately stops further execution of the handleSubmit function, preventing the user from being redirected or falsely shown a successful registration message.
- This condition was crucial in solving the admin registration issue, ensuring that failed registration attempts are clearly communicated when a third user tries to register.
- Without this check, the frontend continued with the registration flow even after backend rejection, creating confusing and misleading feedback, that is; initially for me.

DATABASE RULE – PROBLEM CODE EXPLAINED

AUTHCONTROLLER

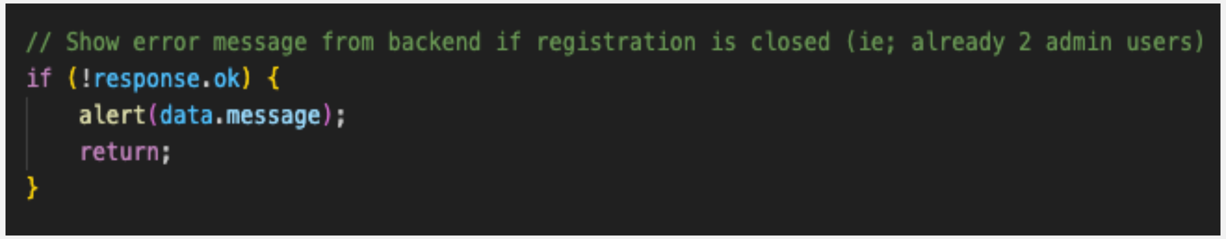


```
// Count the number of registered users in the database (users collection)
const userCount = await UserModel.countDocuments();

// Allow only the first two users to register, and be admins.
if (userCount >= 2) {
    return response
    .status(403)
    .json({ message: "Registration is closed. No new users can be created." });
}
```

- This database rule appears in my backend authController.js file, inside the registerUser function. It uses a conditional check to enforce the rule that only two admin users can register.
- I used countDocuments() from Mongoose to count how many users currently exist in the users collection in my MongoDB atlas database, and saved the count in a variable called userCount.
- My aim was to only allow the first two users to register, and for those users to become admins. After the two admin accounts were created, registration would then be closed.
- The condition if (userCount >= 2) is used to check this. If the count is 2 or more, the backend sends back a 403 Forbidden status and my custom message.
- Even though I had written this logic correctly for the backend, I finally realised that the frontend was not checking the backend's response status.
- So even when registration was blocked, and no new user was added to the database, users still saw a message saying their registration was successful.
- So to fix this, I added a conditional in the frontend (if (!response.ok)), which I explained in the last slide, to first check if the backend had rejected the request.

FUNCTION CALL – PROBLEM CODE EXPLAINED

LOGIN PAGE



```
// Show error message from backend if registration is closed (ie; already 2 admin users)
if (!response.ok) {
    alert(data.message);
    return;
}
```

- This slide focuses on the alert() function, which is called when a user tries to register an admin account and the request fails.
- The alert(data.message) function call is a built in JavaScript function that creates a popup banner to display a message returned from the backend (which I customly set).
- The alert function is triggered inside the handleSubmit function only if the response from the backend fails the condition if (!response.ok).
- This function call played a key role in the admin registration issue, where it seemed that more than 2 users could register.
- When a third user tried to register, the backend correctly returned an error (due to the userCount >= 2 rule), but initially, this message wasn't reaching or alerting the user.
- By adding the function call alert(data.message), I was able to clearly inform users that "Registration is closed," improving their experience by alerting them to the real issue.
- Without this function call, the error message would remain hidden, causing confusion and making it seem like registration was successful when it wasn't.

SUMMARY

KEY CODE – CHECKOUT

- I chose to focus on the handleOrderSubmit function from the Checkout Page because it combines several important parts of my app's logic.
- It includes a conditional to stop the form submitting if any shipping field is empty or the cart has no items, with .trim() used to catch empty spaces.
- The .getTotal() method is called from my custom CalculateOrder class, which uses a for...of **loop** to work out the total price of the cart.
- A network request is then made using fetch() with a POST method to send the order to the backend route /api/orders.
- If the request works, the response is saved to localStorage, a custom toast message is shown with showCartToast(), and the user is redirected with navigate("/success").
- This one function brings together multiple core concepts used throughout my code, including a network request, conditional logic, a loop, and function calls; all working together to successfully submit a customer's order.

PROBLEM CODE – REGISTRATION/LOGIN

- A problem occurred when the frontend did not check the response from the backend fetch request to /api/auth/register.
- This caused a false success message to appear, even when the backend had correctly blocked the registration due to a 2 admin limit I had set.
- The issue involved a network request (fetch), a database rule that was ignored, a missing conditional (if (!response.ok)), and the absence of a function call (alert()) to notify the user.
- On the backend, I had used countDocuments() to limit registrations and return a custom error if the admin limit had been reached.
- Once I added the missing conditional check and alert() function call in the frontend, users were properly informed that registration was in fact closed.
- This issue brought together a network request, a frontend conditional, a backend database rule, and a function call, and solving it really helped me to understand how all of these parts need to work together between the frontend and backend so that my app behaved the way I had expected it to.

FUTURE IMPROVEMENTS

- In future, the seller will be able to upload new products and video tutorials directly through the admin portal.
- Right now, the Checkout and Order Success pages are proxies — later on I'll integrate Shopify APIs to handle real cart creation, checkout, and payment.
- The kit images and videos currently on the site are just placeholders.
- When the site launches properly, we'll upload the real kits, with proper photos, real tutorials, and the full product descriptions
- It is important to note that the seller is still in the process of creating those real kits and videos, so for now I just used example content provided by the seller for this assignment.
- Most of the functionality is already set up, so it'll be easy to update everything once the final content is ready.
- However, what I have built so far is a good representation of what the actual live site will look and feel like.

THANK YOU

By Laylah De Paull

GCAS022002

Web App Link