Entertainment Guild - Web Commerce Application Documentation

Table of Contents

Introduction

The **Entertainment Guild** web commerce application is designed for users to browse, view and purchase a wide variety of books, movies and games for them to enjoy. This documentation serves as a comprehensive guide for developers to understand the structure, functionality and implementation of the system.

Features

User authentication and profile management

User authentication is handled within the system by comparing a password hash against a hashed inputted password.

Customers can signin using the normal login portal to access their profile information, including basic details updating, viewing their order history and deleting their account.

Employees can use the employee login page, and will have access to different panels depending on whether or not the associated use is an administrator or regular employee.

- Employees will be able to view all users and products in the system

- Adminstrators will be able to view, edit and delete all of the details of all items and users within the system, including retaining the ability to grant or revoke administrator privileges for other employees, excluding themselves

Product catalog with detailed descriptions and previews

A search bar and category selection bar is visible at all times during the customer experience. Entering a quote in the search bar or clicking on a category will bring the customer to the **Search page**. From here, customers can scroll through a page of products, filter by category, subcategory and by query.

If a customer wishes to find more information about a product, they can click the button below the product card, and be brought to the **Product Display page**. This will display most of the product's information, including:

- Name, Author, Category and Subcategory
- Price
- The ability to choose their variation and quantity
- Product Description and SKU
- A customer, if they wish, can add the product to their cart

Shopping cart functionality

The contents of a cart is stored in the local storage and tracked as per the contents of the stored information. This allows the cart to be tracked across pages and even when the website is closed down and reopened later.

All cart functions are handled in a Cart Context, wrapped around the entire app in the `App.jsx`. This allows any component to import the context into the component, set the context via the `useContext` function in React and then access all of the functions required for the cart to operate.

Checkout process with order history tracking

Customers are not able to access the checkout screen while their cart is empty. Once products have been added to their cart, customers can proceed to checkout, which gives them a form to fill out for their personal information, billing information and shipping information.

Customers can view their cart in this page as well, as well as cost subtotal and grand total. Upon the form being filled out, the customer can checkout, and the system generates a new order in the database with their information.

Technologies Used

- **Frontend:** React+Vite, React Router
- **State Management:** Redux
- **Styling:** CSS Modules, MaterialUI, React Icons (Font Awesome 5)
- **Backend:** Axios, NocoDB

Overview

- **Client:** The web app serves as the client interface for users to interact with the product catalog.
- **Server:** Handles API requests and business logic.
- **Database:** Stores user data, product details, and order history.

Folder Structure

Folder Structure Overview

├──public │ ├──product │ └──promos └──src ├──assets ├──components │ ├──AdminComponents │ ├──EmployeeComponents │ ├──Profile Components │ └──stylesheets ├──helpers └──pages

Folder Contents

root

Description: The root directory for the project, containing all essential project files and subdirectories.

Content Overview: Contains configuration files, documentation, and important metadata about the project.

public

Description: Contains publicly accessible assets that are served directly to the client, such as images and promotional materials.

Content Overview: Primarily includes promotional and product content

product

Description: Holds product images and related media for the application, ensuring that all visual content for products is organized.

Content Overview: Includes various product images that are utilized throughout the application for display in listings and detail views.

promos

Description: Contains promotional images and marketing assets used throughout the application to enhance user engagement.

Content Overview: Comprises marketing materials designed to attract users, including banners and promotional graphics.

src

Description: The source directory for the application, containing all the code and components necessary for building the app.

Content Overview: Contains JavaScript files, component definitions, and style sheets needed for application functionality.

assets

Description: A collection of static assets such as images, logos, and fonts used throughout the application.

Content Overview: Contains the site logo

components

Description: Contains React components organized into subdirectories for better maintainability and separation of concerns.

Content Overview: Includes reusable UI components like buttons, models, headers, and any other interface elements.

AdminComponents

Description: Components specifically designed for the admin interface, allowing administrative users to manage and edit various aspects of the application.

Content Overview: Houses components that facilitate administrative tasks, including user management, item management and administrator privilege management.

Profile Components

Description: Components used in user profiles, enabling users to view and manage their personal information and settings.

Content Overview: Contains components for displaying and editing user information, order history and account management.

EmployeeComponents

Description: Components used in employee profiles, allowing employees to do their tasks.

Content Overview: Contains components for displaying all user and item information, but not allowing them to edit it.

stylesheets

Description: Contains CSS files and stylesheets that define the visual presentation of the application.

Content Overview: Typically includes stylesheets for main application styling and component-specific styles.

helpers

Description: A collection of utility functions and helper modules that provide reusable logic throughout the application.

Content Overview: Includes various functions for API calls, data formatting, validation, and any other common utilities that will be used app-wide.

pages

Description: Contains main page components that represent different views or routes in the application, facilitating navigation.

Content Overview: Comprises page components that manage the layout and data for various sections of the application.

Setup Instructions

Prerequisites

Before installation, ensure you have the following installed:

- Node.js (v20.18.0 or later)
- npm (installed with Node.js, at least v10.8.2 or later)

Installation Process

1. **Clone the Repository** Open your terminal and clone the GitHub repository. Replace `<repository-url>` with the actual URL of the repository.
   `git clone <repository-url>`

   Navigate to the cloned repository
   `cd <repository-name>`

2. **Install Dependencies** Using `npm`, you can install all of the dependencies through the `npm install` command.

   Note: It's advised that even if you have `node modules` present in this folder, you should remove it and run the dependency installation anyways, as this could lead to `unrecognised` `command` errors.

3. **Running the Project** With your terminal open in the root directory of the repository, using the `npm run dev` command to run the project locally on your machine. This *should* launch the project on http://localhost:5173

Component Documentation

`CartView.jsx`

1. Component Overview

- **Purpose:**
  - The `CartView` component displays the contents of the shopping cart, allowing users to review their selected items, including product details, quantity, price, and total cost. It also provides functionality to remove items from the cart.
- **Usage Context:**
  - This component is used in the checkout process to present a summary of the user's selected products before finalising their order.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It uses context for cart data.

---

3. Return

- **Render Method:**
  - The `CartView` renders a header with column titles, lists the products in the cart with their details, and handles empty cart scenarios.

---

4. Examples

Basic Usage

```
<CartView />
```

---

5. Styling

- **CSS Classes:**
  - Inline styles are used for layout and styling, such as grid layouts for product listings and buttons.
- **Styling Method:**
  - Inline styles are utilized directly within the component.

---

6. Interacting Components

- `CartContext` : Provides the cart data and functions to manipulate the cart (clear and remove items).
- `getProductDetails` : API helper function to fetch product details based on the product ID.
- `Line` : A presentational component used to visually separate sections of the cart.

`CategoryBar.jsx`

1. Component Overview

- **Purpose:**
  - The `CategoryBar` component provides a navigation bar for selecting different product categories.
- **Usage Context:**
  - This component is typically used in the search area of the application to help users quickly access different types of products.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly.

---

3. Return Method

- **Render Method:**
  - The `CategoryBar` renders a list of categories that users can click on to navigate to different product search results.

---

4. Examples

Basic Usage

```
<CategoryBar />---
```

---

5. Styling

- **CSS Classes:**
  - The `cat` class is used for styling the overall container of the category bar.
- **Styling Method:**
  - CSS styles are imported from an external stylesheet (`CategoryBar.css`).

---

6. Interacting Components

- **`CustomLink`** : A component used to create navigation links to different routes within the application.

`CheckoutForm.jsx`

1. Component Overview

- **Purpose:**
  - The `CheckoutForm` component is designed to collect user information when checking out, including contact details and payment information.
- **Usage Context:**
  - This component is used in the checkout page of the application.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`validateForm`	func	Yes	N/A	A callback function to validate the form data and ensure all fields are filled.
`onFormChange`	func	No	N/A	A callback function to pass the updated form data back to the parent component.

---

3. Return Method

- **Render Method:**
  - The `CheckoutForm` renders a series of input fields organized into sections for contact information and payment details. The return statement contains various input elements for user data collection, which can be viewed directly in the `CheckoutForm.jsx` file for inspection.

---

4. Examples

Basic Usage

```
<CategoryBar />
```

---

5. Styling

- **CSS Classes:**
  - The component uses various inline styles and Material-UI components for consistent styling.
- **Styling Method:**
  - Styles are applied using Material-UI's `sx` prop and standard CSS conventions.

---

6. Interacting Components

- **`Grid`** : A Material-UI component used for layout management.
- **`TextField`** : A Material-UI component for input fields.
- **`Typography`** : A Material-UI component for displaying text.
- **`MenuItem`** : A Material-UI component used within dropdowns for selections.

`CostBreakdown.jsx`

1. Component Overview

- **Purpose:**
  - The `CostBreakdown` component provides a detailed view of the cost breakdown for a purchase, including product total, surcharge, and subtotal.
- **Usage Context:**
  - This component is used in the cart and checkout area to help users understand the breakdown of their grand total

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`subtotal`	number	Yes	N/A	The subtotal cost of the items in the cart, used to calculate additional charges.

---

3. Return Method

- **Render Method:**
  - The `CostBreakdown` component renders a styled `div` containing rows that break down costs into product total, surcharge, and subtotal. Each section is displayed with labels and corresponding values aligned side-by-side.

---

4. Examples

Basic Usage

```
<CostBreakdown subtotal={100} />
```

---

5. Styling

- **CSS Classes:**
  - `cost-breakdown`: A main container class for overall component styling.
- **Styling Method:**

    - Inline styling is used throughout, particularly for layout (flexbox) and font sizes. A separate `Line` component is imported and rendered to visually separate sections.

---

6. Interacting Components

- `Line`: A component used to visually separate sections in the cost breakdown.

`CustomLink.jsx`

1. Component Overview

- **Purpose:**
    - The `CustomLink` component creates a styled navigation link that highlights when active.
- **Usage Context:**
    - This component is typically used to make page redirection easier in development when interacting with styling and components that affect navigation.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`to`	string	Yes	N/A	The destination path for the link.
`children`	node	Yes	N/A	The content or elements displayed as the link text.
`props`	object	No	N/A	Additional props passed down to the `Link` component.

---

3. Return Method

- **Render Method:**
    - The `CustomLink` component returns a `Link` element, which includes:
        - Conditional styling that applies an "active" class when the link's path matches the current route.
        - Support for navigation to the specified route when clicked.

---

4. Examples

Basic Usage

```
<CustomLink to="/home">Home</CustomLink>
```

Advanced Usage

```
<CustomLink to="/profile" style={{ color: "blue" }}>Profile</CustomLink>
```

5. Styling

- **CSS Classes:**
  - `active` : This class is applied to the link when the current route matches the `to` prop, allowing for active state styling.
- **Styling Method:**
  - Styling is applied conditionally based on the link's active state, with any additional styles passed through `props` .

---

6. Interacting Components

- `Link` : Imported from `react-router-dom` , it enables client-side navigation.
- `useMatch` and `useResolvedPath` : Hooks that determine if the link's path matches the current route.
- `useNavigate` : This hook is available for future expansion if direct navigation functionality is needed.

`FilterBox.jsx`

1. Component Overview

- **Purpose:**
  - The `FilterBox` component allows users to filter products based on categories and subcategories. It dynamically updates the URL parameters with the current filter selections to support search functionality.
- **Usage Context:**
  - Used throughout the user experience to refine search parameters to locate their ideal product

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive any props directly.

3. Return Method

- **Render Method**
  - The `FilterBox` component renders a form with two main sections: `Categories` and `Subcategories`.
    - It fetches subcategories based on the selected category, and it updates the URL parameters to reflect the selected filters.

4. Examples

Basic Usage

```
<FilterBox />
```

5. Styling

- **CSS Classes:**

  - The component uses external styling from the FilterBox.css stylesheet. Classes include .top-inside for container styling and .filters-container for filter grouping.

6. Interacting Components

- `getFromGenre`: A helper function that fetches subcategory data based on the selected category.
- `useNavigate` and `useLocation`: Hooks used to update URL parameters and support navigation based on filter selections.
- `useSearchParams`: React Router hook used to retrieve URL search parameters for initializing category and search term.

`FilterRadio.jsx`

1. Component Overview

- **Purpose:**
  - The `FilterRadio` component displays an individual radio button with a label, intended to work as part of a group of radio buttons for selecting options in a filter or form.
- **Usage Context:**
  - This component is used in filter sections or forms where users select a single option from multiple radio buttons.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`label`	String	Yes	N/A	The text label displayed alongside the radio button.
`value`	String	Yes	N/A	The value assigned to this radio button option.
`checked`	Bool	Yes	N/A	Indicates whether the radio button is selected.
`onChange`	Func	Yes	N/A	Event handler function that triggers when the selection changes.

---

3. Return Method

- **Render Method**
  - Returns a labeled radio button wrapped in a `FormControlLabel`, using Material UI components for integration into a filter or form.

---

4. Examples

Basic Usage

```
<FilterRadio
  label="Books"
  value="books"
  checked={selectedValue === "books"}
  onChange={handleCategoryChange}
/>```
```

5. Styling

- **CSS Classes:**
  - This component uses Material UI's built-in styling through FormControlLabel and Radio, without custom CSS.

---

6. Interacting Components

- **Parent Filter Components:** Returns values up the chain to its parent component for use in radio button groups

---

`FormFill.jsx`

1. Component Overview

- **Purpose:**
  - The `FormFill` component dynamically generates a form with input fields based on provided placeholder text. It allows conditional rendering of password fields and tracks input changes.
- **Usage Context:**
  - Used in user registration, login forms and other scenarios requiring such inputs

---

2. Props/Attributes

Prop Name	**Type**	**Required**	**Default Value**	**Description**
`placeholders`	Array	Yes	N/A	An array of placeholder strings, one for each input field to be displayed.
`passwordIndexes`	Array	Yes	N/A	An array of indexes indicating which input fields should be of type `password`.
`onFormChange`	Function	Yes	N/A	Callback function triggered when any input value changes, providing the latest form data.

---

3. Return Method

- **Render Method**

  - Renders a series of input fields within a `form-container` div. Each field displays a placeholder and conditionally renders as a password field if its index is in `passwordIndexes`. Values are stored in local state and passed up via `onFormChange`.

---

4. Examples

Basic Usage

```
<FormFill
placeholders={["Username", "Password", "Email"]}
 passwordIndexes={[1]}
 onFormChange={(updatedValues) => console.log(updatedValues)}
/>
```

5. Styling

- **CSS Classes:**
  - `form-container`: Styles the overall container for the form inputs CSS.
  - `input-field`: Wraps each input element for additional styling
- **Styling Method:**
  - Custom styles are applied via the `FormFill.css` stylesheet

---

6. Interacting Components

- **Parent Component Callback:** Receives form data updates through the onFormChange callback, enabling real-time data handling and validation.

`ImageSlider.jsx`

1. Component Overview

- **Purpose:**
  - The `ImageSlider` component displays an image carousel with navigation arrows for moving through a series of images, as well as navigation dots to jump to specific slides.
- **Usage Context:**
  - Used in home page to display promotional content

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`slides`	Array	Yes	N/A	An array of slide objects, each containing a `url` property for an image.

---

3. Return Method

- **Render Method**
  - The component renders an image slider with navigation arrows on each side, a clickable dot navigation below the image, and a slide background set to the current image URL. `slideStyle` is used to style the current image dynamically as the slide background.`.

---

4. Examples

Basic Usage

```
const slides = [
 { url: "path/to/image1.jpg" },
 { url: "path/to/image2.jpg" },
 { url: "path/to/image3.jpg" }
];
```

5. Styling

- **CSS Classes:**
  - `container` : The main wrapper for the image slider component.
  - `slider` : Wraps the image, arrows, and dots for styling.
  - `left-arrow` and `right-arrow` : Style the navigation arrows for slide transitions.
  - `dot-container` : Contains the clickable navigation dots.
  - `dot` : Individual clickable dots for slide selection.
- **Styling Method:**
  - Custom styles are applied via the `ImageSlider.css` stylesheet

---

6. Interacting Components

- None

---

entation

1. Component Overview

- **Purpose:**
  - The `Line` component serves as a styled horizontal line.
- **Usage Context:**
  - Used to separate content for easy visual grouping.

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not accept any props.

---

3. Return Method

- **Render Method**
  - Returns a simple stylised `<hr>` element

---

4. Examples

Basic Usage

```
<Line />
```

5. Styling

- **CSS Classes:**
  - No classes are used as this component exclusively uses inline styling
- **Styling Method:**
  - Inline styling is used
    - `border` : Set to none to remove the default border.
    - `background` : Set to none to avoid any default background color.
    - `boxShadow` : Provides a shadow effect to create a colored line.
    - `marginTop` : Adds spacing above the line.
    - `height` : Defines the thickness of the line (adjustable).

        - `width` : Defines the width of the line.

---

6. Interacting Components

- None

`ProductCard.jsx`

1. Component Overview

- **Purpose:**
    - The `ProductCard` component displays product information in a quickview card format
- **Usage Context:**
    - Used on the homepage and search page to easily display groups of products without cluttering the screen

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`imgSrc`	`string`	No	`"/product/ProductPlaceholder.webp"`	The source URL of the product image.
`imgAlt`	`string`	No	`"Placeholder"`	The alternative text for the product image.
`title`	`string`	Yes	N/A	The title of the product.
`desc`	`string`	No	`"Example Description"`	A brief description of the product.
`buttonText`	`string`	No	`"Check it out!"`	The text displayed on the action button.
`link`	`string`	Yes	N/A	The URL to which the button should link.

---

3. Return Method

- **Render Method**
  - The `ProductCard` renders a structured layout with an image, title, description, and a clickable link. The description text is truncated if it exceeds a specified length to standardise the card height

---

4. Examples

Basic Usage

```
<ProductCard
    title="Example Product"
    link="/product/example"
/>
```

Advanced Styling

```
<ProductCard
    imgSrc="/product/example.jpg"
    imgAlt="Example Product"
    title="Example Product"
    desc="This is a more detailed description of the product that exceeds the
character limit."
    buttonText="View Details"
    link="/product/example"
/>
```

5. Styling

- **CSS Classes:**
  - `card-container` : Master card container
  - `image-container` : Style options for the image, mainly for size
  - `product-details` : Container for text-based components, like title and description
  - `title` : Styling for the card title
  - `link` : Styling for the link button
  - `description` : Styling for the description of the product, smaller than the title
- **Styling Method:**
  - Inline styling is used for specific adjustments, otherwise it relies on the external `ProductCard.css` stylesheet

---

6. Interacting Components

- `CustomLink` : A custom link component used for navigation.

`PromoSection.jsx`

1. Component Overview

- **Purpose:**
  - Main component for displaying promotional content to a user
- **Usage Context:**
  - Used on the home page for displaying the sliding promotional content

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It defines its slides internally.

---

3. Return Method

- **Render Method**
  - The `PromoSection` renders a container that includes an `ImageSlider` component to display the promotional images

---

4. Examples

Basic Usage

```
<PromoSection />
```

5. Styling

- **CSS Classes:**
  - `container` : Master container for the content
- **Styling Method:**

    - Relies on the external `PromoSection.css` stylesheet

---

6. Interacting Components

- `ImageSlider`: This component receives an array of slides and handles the display and transitions between promotional images.

`SearchBar.jsx`

1. Component Overview

- **Purpose:**
    - Allows users to input search queries to navigate the products
- **Usage Context:**
    - Located in the header of the website, available at all times to the user

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It manages its own internal state.

---

3. Return Method

- **Render Method**
    - Renders an input field with a search icon. The search icon is a `CustomLink` component that redirects the user to the search page with the inputted query

---

4. Examples

Basic Usage

```
<SearchBar />
```

5. Styling

- **CSS Classes:**
    - `search-wrapper` : Master container for the content
- **Styling Method:**
    - Relies on the external `SearchBar.css` stylesheet

---

6. Interacting Components

- `CustomLink` : A custom link component used for navigation.
- `FaSearch` : An icon from the react-icons library that visually represents the search functionality.

`StylisedRadio.jsx`

1. Component Overview

- **Purpose:**

    - Customisable radio button interface for rendering different components based on the active selection

- **Usage Context:**

    - Used on the profile and panel pages, such as the user profile, employee panel and admin panel

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`options`	Array	Yes	N/A	An array of strings representing the radio button options.
`iconNames`	Array	No	`[]`	An array of icon names corresponding to each option. Must match the icon names from `react-icons/fa`.
`defaultSelection`	Number	No	`0`	The index of the option that should be selected by default.
`inactiveColour`	String	No	`"#553F16"`	The background color for inactive options.
`activeColour`	String	No	`"#D59C36"`	The background color for the selected option.
`columnStyle`	Boolean	No	`false`	If `true`, options will be displayed in a vertical column.
`onOptionSelect`	Function	No	N/A	A callback function that is called with the index of the selected option when an option is clicked.

---

3. Return Method

- **Render Method**
    - Renders a list of options, with text and an accompanying icon. A different background colour determines the active selection, and updates when clicking on an inactive button

---

4. Examples

Basic Usage

```
<StylisedRadio
 options={['Option 1', 'Option 2', 'Option 3']}
 iconNames={['FaBeer', 'FaCoffee', 'FaApple']}
 onOptionSelect={(index) => console.log(`Selected index: ${index}`)}
/>
```

Advanced Usage

```
<StylisedRadio
 options={['View Items', 'View Accounts', 'Logout']}
 iconNames={["FaClipboardList", "FaUserShield", "FaDoorOpen"]}
 columnStyle={true}
 inactiveColour="#141414"
 activeColour="#553F16"
 onOptionSelect={[
 () => setActiveOption(0),
 () => setActiveOption(1),
 () => handleLogout()
 ]}
/>
```

5. Styling

- **CSS Classes:**
  - No specified classes, utilised some inline styling for handling active and inactive colourings
- **Styling Method:**
  - Relies on the external `StylisedRadio.css` stylesheet

---

6. Interacting Components

- `Box` and `Typography` : These MUI components are used to structure the layout and text formatting within the radio options.

`SubtotalView.jsx`

- **Purpose:**
  - Displays cost information from the cart, including subtotal, surcharge and grand total
- **Usage Context:**
  - Used in the cart and checkout pages for an all-the-time view of the current cart for the user to confirm before buying

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
`onCheckout`	Boolean	No	`false`	If `true` , displays a detailed cost breakdown without the checkout button.

---

3. Return Method

- **Render Method**
  - Conditionally renders either cost breakdown or complete summary with a checkout button, depending on the boolean prop input

---

4. Examples

Basic Usage

```
<SubtotalView onCheckout={true} />
```

5. Styling

- **CSS Classes:**
  - No specified classes, utilised some inline styling for handling active and inactive colourings
- **Styling Method:**
  - Inline styling

---

6. Interacting Components

- `Line` : Simple stylised horizontal line for visual content grouping
- `CartContext` : Utilised for cart data and manipulation
- `CustomLink` : Simple navigation component

`TopBar.jsx`

- **Purpose:**
  - Displays site logo, searchbar and clickable icons for profile and cart
- **Usage Context:**
  - Visible throughout entire user experience for consistency

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It manages its own internal state.

---

3. Return Method

- **Render Method**
  - Renders the topbar containing site logo (link to homepage), search bar and icons for profile and cart navigation

---

4. Examples

Basic Usage

```
<TopBar />
```

5. Styling

- **CSS Classes:**
  - `topbar` : Master container class
  - `search-bar-container` : Wrapper class for positioning the search bar
  - `icon` : Wrapper class for icon styling
- **Styling Method:**
  - Rlies on `TopBar.css` for styling

---

6. Interacting Components

- `SearchBar` : Provides a search functionality for products.
- `CustomLink` : Used for navigation to different routes in the application.

`AdminManagement.jsx`

- **Purpose:**
  - Allows administrators to manage the admin status of other employees

- **Usage Context:**
    - Visible only to adminstrators logged in through the employee portal

---

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
adminKey	string	Yes	N/A	The username of the current admin, used to restrict actions on self.

---

3. Return Method

- **Render Method**
    - Renders a page to view the employees of the company, with a button to grant and revoke administrator privileges

---

4. Examples

Basic Usage

```
<AdminManagement adminKey="currentAdminUsername" />
```

5. Styling

- **CSS Classes:**
    - `topbar` : Master container class
    - `search-bar-container` : Wrapper class for positioning the search bar
    - `icon` : Wrapper class for icon styling
- **Styling Method:**
    - Rlies on `TopBar.css` for styling

---

6. Interacting Components

- `Line` : Used to create a horizontal line separator.

`ItemManagement.jsx`

1. Component Overview

- **Purpose:**
    - Allows administrators to manage their assigned items, including the ability to view, edit, add and delete items from the database.
- **Usage Context:**
    - Visible only to adminstrators logged in through the employee portal

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
adminKey	string	Yes	N/A	The unique key for the admin user, used to fetch and manage items.

3. Return Method

- **Render Method**
    - Renders a page to view their assigned items, allowing editing, viewing, deleting and adding of items.

4. Examples

```
<ItemManagement adminKey="adminKeyValue" />
```

5. Styling

- **CSS Classes:**
    - `item-management` : Main container for the item management component
    - `line` : Visual separator between sections
    - `grid-layout` : Defines the grid structure for displaying items
    - `button` : Standard styling for buttons throughout the component.
    - `input` : Styles for text input fields, including borders and padding
- **Styling Method:**
    - Relies on inline styles for layout and uses a consistent color theme for borders and backgrounds.

6. Interacting Components

- `Line` : Used to create a horizontal line separator.

`UserManagement.jsx`

1. Component Overview

- **Purpose:**
  - Allows administrators to manage patrons in the database, viewing, editing, adding and deleting accounts
- **Usage Context:**
  - Visible only to adminstrators logged in through the employee portal

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
adminKey	string	Yes	N/A	The unique key for the admin user, used to fetch and manage items.

3. Return Method

- **Render Method**
  - Allows the administrator to view the patrons of the company, allowing for viewing, editing, deleting and adding of new users.

4. Examples

```
<UserManagement adminKey="adminKeyValue" />
```

5. Styling

- **CSS Classes:**
  - `user-management` : Main container class for the component.
  - `user-list` : Styles for the grid layout of the user list.
  - `input` : Base styling for input fields across the component.
  - `button` : Styles for the buttons used in the component.
- **Styling Method:**
  - Relies on inline styles for layout and uses a consistent color theme for borders and backgrounds.

6. Interacting Components

- `Line` : Used to create a horizontal line separator.

`ItemView.jsx`

1. Component Overview

- **Purpose:**
  - Allows employees to view all of the items in the system
- **Usage Context:**
  - Visible only to employees logged in through the employee portal

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It manages its own internal state.

3. Return Method

- **Render Method**
  - Renders a table of all products, with buttons to navigate pages of products.

4. Examples

```
<ItemView />
```

5. Styling

- **CSS Classes:**
  - No specific classes were used, just inline style props.
- **Styling Method:**
  - Relies on inline styles for layout and uses a consistent color theme for borders and backgrounds.

6. Interacting Components

- `Line` : Used to create a horizontal line separator.

`UserView.jsx`

1. Component Overview

- **Purpose:**
    - Allows employees to view all of the patrons in the system
- **Usage Context:**
    - Visible only to employees logged in through the employee portal

2. Props/Attributes

Prop Name	Type	Required	Default Value	Description
N/A	N/A	N/A	N/A	This component does not receive props directly. It manages its own internal state.

3. Return Method

- **Render Method**
    - Renders a table of all patrons, with buttons to navigate pages of patrons.

4. Examples

```
<UserView />
```

5. Styling

- **CSS Classes:**
    - No specific classes were used, just inline style props.
- **Styling Method:**
    - Relies on inline styles for layout and uses a consistent color theme for borders and backgrounds.

6. Interacting Components

- `Line` : Used to create a horizontal line separator.

Helper Documentation

AdminRouter

Purpose: Restrict access to routes only for admin use. Non-admin users cannot access admin routes, and admin users cannot access non-admin routes

Functions

AdminRoute

Props

- **isAdmin** (`boolean`): Indicates if user has admin permissions
- **children** (`ReactNode`): Component to render if user has admin access

Usage

```
<AdminRoute isAdmin={user.isAdmin}>
  <AdminPanel />
</AdminRoute>
```

If `isAdmin` is `false` , the user will be redirected to the home page

NonAdminRoute

Props

- **isAdmin** (`boolean`): Indicates if user has admin permissions
- **children** (`ReactNode`): Component to render if user does not have admin access

Usage

```
<NonAdminRoute isAdmin={user.isAdmin}>
  <UserDashboard />
</NonAdminRoute>
```

If `isAdmin` is `true` , the user is automatically redirected to the admin panel

---

CartContext

Purpose: Manage and persist the state of a shopping cart within the application. Cart data is stored in `localStorage` to retain state across page reloads.

Functions

CartProvider

Purpose: Wraps components with `CartContext.Provider`, providing access to the cart state and functions for managing the cart.

State

- **cart** (`array`): The current contents of the cart. Each cart item includes:
  - **productID** (`string`): Unique identifier for the product.
  - **quantity** (`number`): Quantity of the product in the cart.
  - **source** (`string`): Source of the product.

Methods

- **addToCart** (`productID` , `quantity` , `source`):
  - Adds a specified quantity of a product to the cart. If an item with the same `productID` and `source` exists, it increments the quantity; otherwise, it adds a new item.
- **removeFromCart** (`productID` , `source`):
  - Removes a specific product from the cart based on both `productID` and `source` .
- **clearCart** ():
  - Clears all items from the cart.
- **getCart** ():
  - Returns the current cart contents.

Usage

Wrap the application with `CartProvider` to enable any component within it to access the cart state and management functions.

```
<CartProvider>
  <App />
</CartProvider>
```

The cart information can be accessed using the `useContext` function in `'react'`

```
import { useContext } from 'react';
import { CartContext } from './CartContext';

const { cart, addToCart, removeFromCart, clearCart, getCart } =
useContext(CartContext);
```

HandleLogin

Purpose: Provide a set of functions for handling user login, password validation, and session management, including a hashing function for secure password verification.

Functions

---

sha256

Props

- **message** (`string`): message to encrypt

Usage

```
const hash = await sha256("myPassword");
```

HandleLogin

Props

- **username** (`string`): email of patron account
- **password** (`string`): password of patron account

Usage

```
const result = await HandleLogin("user@example.com", "password123");
```

---

employeeLogin

Props

- **username** (`string`): email of patron account
- **password** (`string`): password of patron account

Usage

```
const result = await employeeLogin("employee@example.com", "password123");
```

---

checkPassword

Props

- **userID** (`string`): User ID of patron account
- **password** (`string`): password of patron account

Usage

```
const result = await checkPassword("12345", "password123");
```

---

logout

Props

- **navigate** (`function`): `useNavigate()` object from parent component

Usage

```
logout(navigate);
```

API Documentation

GET API Functions

`getFromGenre()`

Purpose: Fetches a list of genres based on specified category

Parameters:

- **genre** (`string`): Category type (expecting `books` , `movies` , `games`)

Returns: Array of genre names

Usage:

```
const genres = await getFromGenre("books");
```

`getRandomProduct()`

Purpose: Fetches a random product. It retries until it finds a valid product or reaches a maximum retry count.

Returns: Product details or throws an error after reaching max retry attempts.

Usage:

```
const product = await getRandomProduct();
```

`searchProducts(options)`

Purpose: Searches for products based on query, category, subcategories, and pageoptions.

Parameters:

- **options** (`object`):
  - **query** (`string`): Search query.
  - **category** (`string`): Main category.
  - **subcategories** (`array`): List of subcategory IDs.
  - **currentPage** (`number`): Current page.

Returns: Array of product IDs and total row count for page generation.

Usage:

```
const [productIDs, totalRows] = await searchProducts({ query: "Adventure", category:
"books" });
```

`getProductDetails(productID)`

Purpose: Fetches detailed information for a specified product, including price and stock quantity.

Parameters:

- **productID** (`number`): Unique product ID.

Returns: Object containing product and stock details.

Usage:

```
const productDetails = await getProductDetails(1);
```

`getCategoryMappings()`

Purpose: Provides a mapping of categories to their respective IDs.

Returns: Object with category names as keys and IDs as values.

Usage:

```
const categoryMap = getCategoryMappings();
```

`getSubcategoryMappings(category)`

Purpose: Fetches a list of subcategories for a specified category.

Parameters:

- **category** (`string`): Category type.

Returns: Object mapping subcategory names to their IDs.

Usage:

```
const subcategories = await getSubcategoryMappings("books");
```

`getOrders(userID)`

Purpose: Fetches all orders associated with a specific user.

Parameters:

- **userID** (`number`): User ID.

Returns: Array of orders.

Usage:

```
const orders = await getOrders(123);
```

`getSourceFromID(sourceList)`

Purpose: Fetches source names for a list of source IDs.

Parameters:

- **sourceList** (`array`): Array of source ID objects.

Returns: Array of source names.

Usage:

```
const sources = await getSourceFromID([{ SourceId: 1 }, { SourceId: 2 }]);
```

`getSourceFromSourceID(id)`

Purpose: Fetches the source name for a single source ID.

Parameters:

- **id** (`number`): Source ID.

Returns: Source name.

Usage:

```
const sourceName = await getSourceFromSourceID(1);
```

`getAdminItems(adminKey)`

Purpose: Fetches items available to an admin, including category, subcategory, and source details.

Parameters:

- **adminKey** (`string`): Admin key for access.

Returns: Array of product details.

Usage:

```
const adminItems = await getAdminItems("administratorOne");
```

`getAllCategoriesAndSubcategories()`

Purpose: Fetches all categories and their respective subcategories.

Returns: Object mapping categories to their subcategories.

Usage:

```
const categories = await getAllCategoriesAndSubcategories();
```

`getAllSources()`

Purpose: Fetches all sources from the API.

Returns: List of all sources.

Usage:

```
const sources = await getAllSources();
```

`getStocktake(itemID)`

Purpose: Fetches stock information for a specified item.

Parameters:

- **itemID** (`number`): Product ID.

Returns: List of stock details.

Usage:

```
const stocktake = await getStocktake(123);
```

`getUsers()`

Purpose: Fetches a list of all registered users.

Returns: List of users.

Usage:

```
const users = await getUsers();
```

`getEmployees()`

Purpose: Fetches a list of all employees.

Returns: List of employees.

Usage:

```
const employees = await getEmployees();
```

`getEmployeeItems(currentPage, itemsPerPage)`

Purpose: Fetches all items with page support.

Parameters:

- **currentPage** (`number`): Current page number.
- **itemsPerPage** (`number`): Number of items per page.

Returns: Array of employee items and page details.

Usage:

```
const employeeItems = await getEmployeeItems(1, 10);
```

POST API Functions

`updateName(userID, newName)`

Purpose: Updates the name of a user identified by their user ID.

Parameters:

- **userID** (`number`): Unique identifier for the user.
- **newName** (`string`): The new name to set for the user.

Returns: None

Usage:

```
updateName(123, "John Doe");
```

`updateEmail(userID, newEmail)`

Purpose: Updates the email of a user identified by their user ID.

Parameters:

- **userID** (`number`): Unique identifier for the user.
- **newEmail** (`string`): The new email to set for the user.

Returns: None

Usage:

```
updateEmail(123, "john.doe@example.com");
```

`updateOrder(orderID, streetAddress, suburb, state, postcode)`

Purpose: Updates the address details of a specified order.

Parameters:

- **orderID** (`number`): Unique identifier for the order.
- **streetAddress** (`string`): New street address.
- **suburb** (`string`): New suburb.
- **state** (`string`): New state.

- **postcode** (`string`): New postcode.

Returns: None

Usage:

```
updateOrder(456, "123 Main St", "Townsville", "State", "12345");
```

`deleteUser(userID)`

Purpose: Deletes a user identified by their user ID.

Parameters:

- **userID** (`number`): Unique identifier for the user.

Returns: None

Usage:

```
deleteUser(123);
```

`addNewItem(data)`

Purpose: Adds a new product to the database.

Parameters:

- **data** (`object`): Object containing product details (e.g., `Name` , `Author` , `Description` , etc.).

Returns: None

Usage:};

```
addNewItem({
    Name: "New Book",
    Author: "Author Name",
    Description: "Description of the book",
    Category: "Fiction",
    Subcategory: "Fantasy",
    Published: "2023-01-01",
    LastUpdatedBy: "admin",
    LastUpdated: "2023-10-26",
    Sources: [{ SourceId: 1, Quantity: 5, Price: 10.99 }]
});
```

`updateStocktakesForNewItem(productId, sources)`

Purpose: Updates stocktakes for a newly added product.

Parameters:

- **productId** (`number`): ID of the newly added product.
- **sources** (`array`): Array of sources for the product with quantity and price.

Returns: None

Usage:

```
updateStocktakesForNewItem(1, [{ SourceId: 1, Quantity: 5, Price: 10.99 }]);
```

`updateItem(data)`

Purpose: Updates an existing product in the database.

Parameters:

- **data** (`object`): Object containing updated product details.

Returns: None

Usage:

```
updateItem({
    ID: 1,
    Name: "Updated Book",
    Author: "Updated Author",
    Description: "Updated description",
    Category: "Non-Fiction",
    Subcategory: "Biography",
    Published: "2022-01-01",
    LastUpdatedBy: "admin",
    LastUpdated: "2023-10-26",
    Sources: [{ SourceId: 1, Quantity: 3, Price: 12.99 }]
});
```

`updateStocktakes(data)`

Purpose: Updates stocktakes for a product based on provided data.

Parameters:

- **data** (`object`): Object containing product ID and sources to update.

Returns: None

Usage:

```
updateStocktakes({
    ID: 1,
    Sources: [{ SourceId: 1, Quantity: 5, Price: 10.99 }]
});
```

`addNewUser(data)`

Purpose: Adds a new user to the database.

Parameters:

- **data** (`object`): Object containing user details (e.g., `Email` , `Name` , `Salt`).

Returns: None

Usage:

```
addNewUser({
    Email: "john.doe@example.com",
    Name: "John Doe",
    Salt: "randomSalt"
});
```

`updateUser(data)`

Purpose: Updates an existing user's information.

Parameters:

- **data** (`object`): Object containing updated user details.

Returns: None

Usage:

```
updateUser({
    UserID: 123,
    Email: "john.doe@example.com",
    Name: "John Doe"
});
```

`updateUserAdminStatus(user)`

Purpose: Updates the admin status of a user.

Parameters:

- **user** (`object`): Object containing user details, including `UserName` and `IsAdmin` status.

Returns: None

Usage:

```
const updatedUser = {
    ...user,
    IsAdmin: isAdminInt === 1 ? 0 : 1, // Toggle admin status using integers
};

updateUserAdminStatus(updatedUser);
```

`deleteItem(itemData)`

Purpose: Deletes a specified product from the database, including its stocktakes.

Parameters:

- **itemData** (`object`): Object containing item details, including the ID.

Returns: Response data from the deletion operation.

Usage:

```
deleteItem({ ID: 1 });
```

`deleteStocktakesByProductId(productId)`

Purpose: Deletes all stocktakes associated with a specified product ID.

Parameters:

- **productId** (`number`): ID of the product whose stocktakes need to be deleted.

Returns: None

Usage:

```
deleteStocktakesByProductId(1);
```

`addNewOrder(data, cart)`

Purpose: Adds a new order to the system, processing customer information and items in the cart.

Parameters:

- **data** (`object`): Object containing order details (e.g., email, address, payment info).
- **cart** (`array`): Array of items being ordered.

Returns: None

Usage:

```
addNewOrder({
    email: "john.doe@example.com",
    phone: "123456789",
    address: "123 Main St",
    postcode: "12345",
    city: "Townsville",
    state: "State",
    cardNumber: "1234567812345678",
    cardName: "John Doe",
    expMonth: "12",
    expYear: "2025",
    cvv: "123"
}, [{ productID: 1, source: "Source Name", quantity: 1 }]);
```

Personal Comments

Notes about Order Functionality

Whilst working on the Orders system, more specifically creating orders, I ran into some issues. When analysing the documentation, there are three tables associated with orders:

- `TO`
- `Orders`
- `ProductsInOrders`

I understood the flow as such:

1. Create a new `TO` to store the customer address and billing information
2. Create a new order linked to that TO's `customerID`
3. Create separate entries for each individual item in the cart in `ProductsInOrders`, using the `orderID` and `ItemId` as the two foreign keys

I was able to create `TO` and `Order` entries just fine, but for some reason, there seemed to be an issue with `ProductsInOrders`, as it did not allow me to perform any calls other than `GET`. `POST`, `PUT`, `PATCH`, none of these seemed to work.

Additionally, the `Stocktake List` and `ProductsInOrders List` that generate alongside the Orders creation refused to update, despite explicitly stating them in the data to update. I investigated the schema, and found out the `Stocktake List` in `Orders` and the `Orders List` in `Stocktake` are a many-to-many relationship, and for some reason, I was unable to update either of these entries to link products to orders.

As a result, when a user checks out and creates a new order, the order WILL appear in their order history but it will not contain any product or pricing information. I did not allocate enough time to try and solve this as this was an unexpected bug and I was unable to fix it.