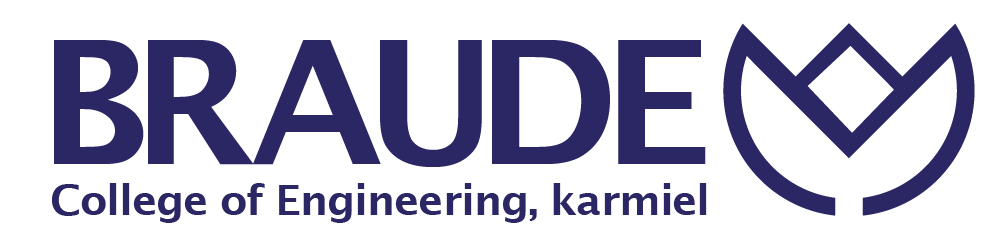
****

**Software Engineering Department**

**Ort Braude College**

**Capstone Project Phase B**

**Buffet Decoration Ordering System**

**Project team members: Razi Dallasheh , Abrar Fauzi**

**Advisor’s name:  Zeev Barzily**

**Table of Contents**

**Abstract…………………………………………………………………………………………………………..3**

**Introduction…………………………………………………………………………………………………….4**

**General Description………………………………………………………………………………..………..5**

**Activity Diagram…………………………………………………………………………………………….6-7**

**Development Process Description…………………………………………………….……………8-10**

**Tools Used and Client Interface During Development……………….…………….10-11**

**Challenges Faced and Solutions………………………………………………………12-14**

**Results and Insights…………………………………………………………………………………...14-15**

**Coping with Challenges……………………………………………….………………………………….16**

**Decision-Making Considerations…………………………………………..………………………….17**

**Lessons Learned………………………………………………………………………………………..18-20**

**Did We Meet the Project Benchmarks…..............................................................20-21**

**User Guide………………………………………………………………………………………….……..22-31**

**Maintenance Guide………………………………………………………………………32-47**

**Abstract**

The Buffet Decoration Ordering System is an innovative solution designed to streamline the process of selecting, customizing, and ordering decorations for buffet events. This system aims to simplify the decoration planning process for event organizers by providing a user-friendly platform where users can browse various decoration options, customize their selections, and efficiently place orders. The system is tailored to meet the needs of both professional event planners and individuals hosting personal events, ensuring a seamless experience from selection to delivery. With its focus on personalization, inventory management, and order tracking, the system addresses the common challenges faced in event decoration planning, offering a reliable and scalable solution that enhances the visual appeal of any buffet setup.

**Introduction**

Event planning, especially when it comes to creating an aesthetically pleasing buffet setup, can be a complex and time-consuming task. The need to coordinate various decorative elements, ensure timely delivery, and stay within budget often presents significant challenges to both professional event organizers and individuals planning personal events. Recognizing these challenges, we have developed the Buffet Decoration Ordering System, a comprehensive tool designed to simplify and enhance the decoration ordering process.

This system allows users to explore a diverse range of decoration options, customize their choices according to their event's theme, and place orders with ease. By integrating key features such as inventory management, order tracking, and secure payment processing, the system ensures that every aspect of the decoration process is handled efficiently and professionally.

The Buffet Decoration Ordering System is designed with the user in mind, offering an intuitive interface that caters to both seasoned event planners and those new to the process. Whether for weddings, corporate events, or private parties, the system provides the flexibility and functionality needed to create stunning buffet presentations that leave a lasting impression.

This project not only aims to reduce the complexity of event decoration planning but also seeks to enhance the overall user experience, making it easier than ever to achieve a visually appealing buffet setup. Through this system, we address a significant gap in the event planning industry, offering a solution that is both practical and innovative.

**General Description**

Our project focuses on developing a Buffet Decoration Ordering System designed to streamline the process of selecting, customizing, and ordering decorations for buffet events. The primary goal of the system is to provide users with an intuitive and efficient way to personalize their buffet setups according to their preferences, event themes, and budget.

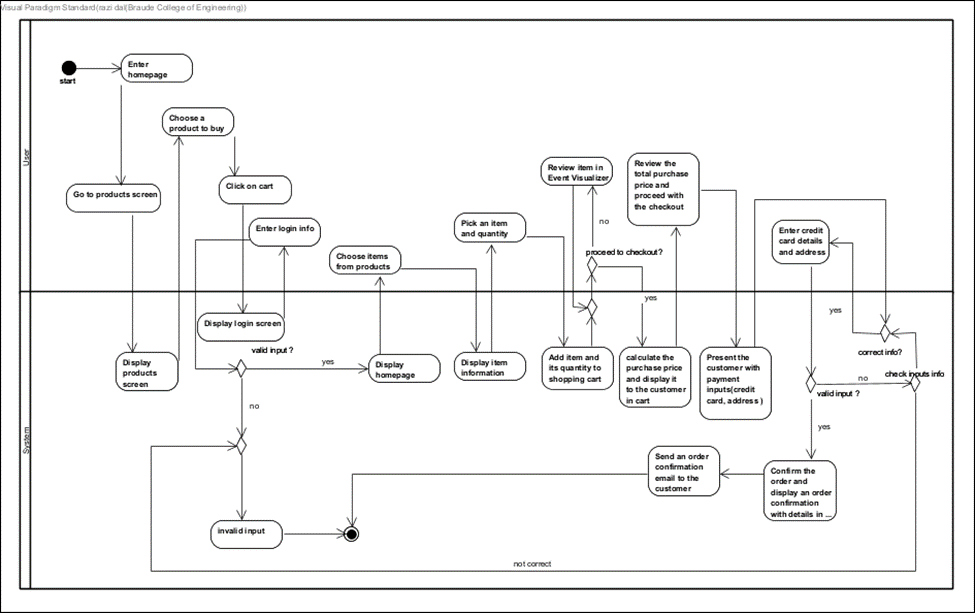
Key Objectives:

* **Simplify the Ordering Process:** The system allows users to browse a wide range of decoration options, customize their choices, and place orders seamlessly.
* **Personalization:** Users can mix and match different decoration elements to create a unique look that aligns with their event's theme.
* **Efficient Management**: The system manages inventory, tracks orders, and schedules deliveries to ensure that everything is ready for the event on time.

**System Architecture:** The system is built on a robust and scalable architecture, ensuring reliability and efficiency. It includes a user-friendly interface, a secure database for storing user data and order information, and an integration with payment gateways to handle transactions.

**Target Audience**: The primary users of this system are event organizers, caterers, and individuals planning events such as weddings, corporate functions, or private parties. The system caters to both professional event planners and casual users who want to enhance the visual appeal of their buffet presentations.

**Activity Diagram**



**1. Start and Homepage Entry**

* The process begins when the user enters the homepage of the platform. From here, the user can choose a product to buy.

**2. Product Selection and Cart**

* The user navigates to the products screen, selects items, and clicks on the cart to review their selection.
* The system displays the selected products, allowing the user to review the items before proceeding further.

**3. Login Requirement**

* If the user attempts to proceed, they must enter login info.
* The system checks the validity of the input. If the input is invalid, the system will display an error, prompting the user to enter correct information.

**4. Review and Proceed to Checkout**

* After a successful login, the user can view their shopping cart. If the user chooses to proceed to checkout, they are presented with the total purchase price.
* The system prompts the user to enter credit card details and address for payment and shipping.

**5. Payment Information Validation**

* The system validates the entered payment information. If the information is incorrect, the system asks for re-entry.
* Once the information is validated, the user can proceed.

**6. Order Confirmation**

* Upon successful payment, the system calculates the total price and displays the final information to the user.
* The system sends an order confirmation email to the user and displays an order confirmation screen with all the order details.

**7. End of Process**

* The process concludes after the order is confirmed, and the user receives the confirmation email.

**Development Process Description:**

**1. Project Planning and Requirements Gathering**

* **Objective**: Create a web application that allows users to browse and purchase buffet decorations, manage their shopping cart, and handle orders, payments, and user authentication.
* **Key Features**:
  + User Registration and Authentication (with Google Firebase)
  + Product Catalog (Buffet Decorations)
  + Shopping Cart Management
  + Order Placement and Payment Handling
  + Admin Reporting and User Management

**2. Technology Stack Selection**

* **Frontend**: React.js
* **Backend**: Node.js with Express.js
* **Database**:
  + Initially considered PostgreSQL.
  + Switched back to MongoDB with Mongoose.
* **Authentication**: with bcrypt method and Google Firebase.
* **Validation**: express-validator.
* **Session Management**: express-session.

**3. Database Design**

* **Models/Collections**:
  + **User**: Handles user registration, authentication, and role management.
  + **Product**: Stores buffet decoration details such as name, price, quantity, etc.
  + **Order**: Tracks orders placed by users, including items, total cost, and status.
  + **Shopping Cart**: Temporary storage of user-selected products before purchase.
* **MongoDB Implementation**:
  + Used Mongoose for schema definitions and validations.

**4. Initial Setup**

* **Backend Setup**:
  + Initialized Node.js project with necessary packages: Express, Mongoose, bcrypt, dotenv.
  + Configured MongoDB connection using Mongoose.
  + Set up the folder structure: models, controllers, routes.
  + Created environment variables to manage sensitive information (e.g., MongoDB URI, session secret).

**Frontend Setup**:

* Initialized React app using Create React App.
* Set up basic routing using React Router.

**5. Feature Implementation**

* **User Authentication**:
  + Developed controller setup to manage user registration and login.
  + Implemented bcrypt password hashing method in order to hash passwords.
  + Integrated Google Firebase for social login.
  + Created protected routes for users based on their authentication status.
* **Product Management**:
  + Developed a product controller to handle CRUD operations for buffet decorations.
  + Implemented product listings and individual product details on the frontend.
* **Shopping Cart and Order Management**:
  + Created order and shopping cart controllers.
  + Enabled users to add items to the cart, view cart contents, and place orders.
  + Integrated payment processing (mocked/stubbed in development).
* **Reporting**:
  + Developed controller to generate admin reports on user activities and orders.
  + Stored reports in the ReportGenerator collection.

**Tools Used and Client Interface During Development:**

**Frontend (React-based):**

**React Components**: We break down our user interface into reusable and manageable components. Each component encapsulates its logic and presentation, promoting code reusability and maintainability.

**Routing**: We used React Router to define the routes of our application and handle navigation between different views. It allows us to map URLs to React components, enabling seamless navigation and deep linking within our application.

**Styling:** We choose a styling approach that suits our project requirements. CSS provides flexibility and familiarity, while CSS Modules offer local scoping of styles to prevent conflicts. Styled Components allow you to write CSS directly within our JavaScript code, promoting component-based styling.

**Backend (Node.js-based):**

**Express.js:** Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies the process of creating RESTful APIs and handling HTTP requests.

**Routing:** We define routes in our Express application to handle different types of requests. Express Router allows us to organize our route handlers into modular and reusable units, improving code organization and maintainability.

**Authentication and Authorization:** We are implementing authentication and authorization mechanisms to protect our application's resources and enforce access control policies. And we used "bcrypt" for stateless authentication or session-based authentication for server-side session management.

**Database Integration:** We choose a database solution that best fits our application's needs. MongoDB is a NoSQL document database that offers flexibility and scalability.

**Database:**

**MongoDB**: MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like documents. It's suitable for applications with rapidly changing schema or unstructured data.

**Deployment:**

For the deployment of your project, we used two platforms: **Render** for the backend and **Netlify** for the frontend. Here’s how the deployment works:

1. **Backend (Render)**:
   * We deployed the backend using Render, a cloud platform that automatically builds and deploys the app from the connected Git repository.
   * Every time we make a change and push a commit to the Git repository, Render detects the update, automatically rebuilds the backend, and deploys the latest version.
   * This ensures that the backend is always up to date with the latest changes.
2. **Frontend (Netlify)**:
   * For the frontend, we used Netlify, which allows us to deploy static websites or single-page applications (SPA).
   * Similar to Render, Netlify is connected to our Git repository. Whenever a commit is pushed, Netlify automatically rebuilds and deploys the latest version of the frontend.
   * Netlify also provides seamless integration with custom domains and SSL certificates for secure access to the frontend.

This setup ensures that both the backend and frontend are always in sync with the latest code updates and provide continuous integration and deployment (CI/CD) for your project.

**Challenges Faced and Solutions**

**1.Choosing the Right Database Challenge**: Initially, PostgreSQL was chosen as the database. However, as the project requirements evolved, especially with the need for flexible data structures for products and user data, it became clear that a NoSQL database might be a better fit.

* **Solution**: After careful consideration, the decision was made to switch to MongoDB, which offers greater flexibility with schema design. This switch allowed for easier implementation of complex, nested data structures, particularly for products and user information.

**2. User Authentication Complexity**

* **Challenge**: Implementing a robust authentication system with both local and Google Firebase strategies presented challenges in ensuring security and seamless user experience.
* **Solution**: Local strategy was secured with bcrypt for hashing passwords, and Firebase was integrated using Google’s API for social login. This provided a secure and user-friendly authentication process.

**3. Handling Data Validation**

* **Challenge**: Ensuring that user inputs, especially during registration and product management, were properly validated to prevent invalid or malicious data from entering the system.
* **Solution**: Implemented express-validator for server-side validation. This middleware was used to enforce strict rules on inputs, such as email formats and password strength, ensuring that only valid data could be processed and stored.

**4. Managing State and Session Persistence**

* **Challenge**: Ensuring that user sessions were securely managed, especially with a combination of local and Firebase logins.
* **Solution**: Used express-session to manage user sessions on the server side. Sessions were securely stored with proper encryption, and were used to serialize and deserialize user sessions, maintaining persistence across different authentication methods.

**5. Integration of Frontend and Backend**

* **Challenge**: Coordinating the interaction between the React frontend and the Express/MongoDB backend was complex, especially with asynchronous operations such as user registration and login.
* **Solution**: Developed clear and consistent API endpoints in the backend to handle requests from the React frontend. Used Axios in React to handle HTTP requests, and ensured proper handling of promises and asynchronous operations to provide smooth user experience.

**6.Handling Asynchronous Operations**

* **Challenge**: Asynchronous operations, particularly in authentication and database interactions, needed careful handling to avoid race conditions and ensure data integrity.
* **Solution**: Employed async/await syntax in JavaScript to manage asynchronous operations more effectively. This approach allowed for cleaner, more readable code and ensured that operations like user registration, login, and data retrieval were handled sequentially and correctly.

**7.Error Handling and User Feedback**

* **Challenge**: Providing meaningful error messages and feedback to users when something went wrong during processes like registration or login was crucial but challenging.
* **Solution**: Implemented comprehensive error handling both on the server and client sides. Used try/catch blocks in async functions to catch errors and send appropriate responses to the frontend. On the frontend, used state management to display user-friendly error messages and guide the user on corrective actions.

**8.Security Considerations**

* **Challenge**: Ensuring the application was secure against common vulnerabilities like SQL injection, XSS, and data breaches.
* **Solution**: Adopted security best practices such as:
  + Using parameterized queries and input validation to prevent SQL injection.
  + Sanitizing user inputs and outputs to prevent XSS attacks.
  + Storing passwords securely using bcrypt hashing.
  + Implementing HTTPS for secure data transmission.
  + Using environment variables to store sensitive information like database credentials and session secrets.

**9.Optimizing Performance**

* **Challenge**: As the application grew, there was a need to ensure it remained performant, especially with increasing user data and transactions.
* **Solution**: Performed optimizations such as:
  + Indexing MongoDB collections for faster queries.
  + Implementing pagination for data-intensive pages.
  + Using React's lazy loading to load components only when needed, improving the application's load time and responsiveness.

**10.Deployment and Environment Configuration**

* **Challenge**: Deploying the application in a production environment while managing different configurations for development, testing, and production stages.

**Solution**: Utilized environment variables for different stages of development and production. Configured MongoDB Atlas for production use and ensured secure connections with proper SSL/TLS certificates. Used a cloud service (e.g., Heroku, AWS) for deployment, with CI/CD pipelines set up for continuous integration and delivery

**Results and Insights**

**Project Goals Achievement**

The primary goals of the project were to create a fully functional web application for managing buffet decorations, with features including user authentication, product management, and report generation. The project successfully met these goals through careful planning, iterative development, and the use of appropriate technologies. Here's a breakdown of how the goals were achieved:

1. **User Authentication and Security**:
   * **Achievement**: Implemented a robust authentication system that supported both local login and Google Firebase. This ensured that users could securely register and log in to the application, with their passwords securely hashed using bcrypt.
   * **Reasoning**: Security and ease of use were top priorities. By integrating Google Firebase, we provided users with a convenient option to log in using their existing Google accounts, reducing the friction of account creation while maintaining security.
2. **Product Management**:
   * **Achievement**: Developed a backend with MongoDB that allowed for flexible and scalable management of buffet decorations and related products. The frontend was designed with React to provide an intuitive interface for adding, editing, and viewing products.
   * **Reasoning**: MongoDB's schema flexibility was crucial for handling the diverse and dynamic nature of product data, which could include various attributes such as name, price, and quantity. This flexibility allowed the application to adapt to changing business requirements without significant rework.
3. **Report Generation**:
   * **Achievement**: Implemented a reporting system that allowed users to generate and view reports related to product usage and inventory. These reports were stored and managed effectively using MongoDB.
   * **Reasoning**: The ability to generate and manage reports was essential for business insights and decision-making. MongoDB's ability to handle large datasets and complex queries made it an ideal choice for storing and retrieving report data efficiently.
4. **Seamless User Experience**:
   * **Achievement**: The React frontend was designed with a focus on user experience, ensuring that the application was responsive, intuitive, and easy to navigate. Key features such as user registration and product management were streamlined to provide a seamless experience.
   * **Reasoning**: User experience was a critical consideration throughout the project. By focusing on intuitive design and responsiveness, we ensured that users could easily interact with the application, leading to higher user satisfaction and engagement.

**Coping with Challenges**

The project faced several challenges, from choosing the right technology stack to handling complex asynchronous operations. Here's how we coped with these challenges:

1. **Technology Stack**:
   * **Challenge**: Selecting the appropriate database (PostgreSQL vs. MongoDB) and managing the switch when requirements changed.
   * **Coping Strategy**: We conducted a thorough analysis of the project's requirements and determined that MongoDB's flexibility was better suited for the dynamic nature of the data. The transition was managed carefully, ensuring that no data was lost and that the application continued to function seamlessly.
2. **Security and Authentication**:
   * **Challenge**: Implementing a secure authentication system that supports both local and Google logins.
   * **Coping Strategy**: We used methods to manage multiple authentication strategies securely. By using bcrypt for password hashing and securely managing sessions with express-session, we ensured that the authentication system was both secure and user-friendly.
3. **Performance Optimization**:
   * **Challenge**: Ensuring that the application remained performant as it grew in complexity and user base.
   * **Coping Strategy**: We implemented performance optimizations such as indexing MongoDB collections, using pagination for data-heavy operations, and leveraging React's lazy loading. These measures helped maintain the application's responsiveness and efficiency.
4. **Error Handling and User Feedback**:
   * **Challenge**: Providing clear and actionable feedback to users in case of errors.
   * **Coping Strategy**: We implemented comprehensive error handling on both the server and client sides, ensuring that users received meaningful feedback and guidance on how to proceed in case of errors.

**Decision-Making Considerations**

Throughout the project, several key considerations influenced our decision-making:

1. **Scalability and Flexibility**:
   * We prioritized technologies and architectures that allowed for easy scaling and flexibility. This was a major factor in choosing MongoDB over PostgreSQL, as well as in designing the frontend with React.
2. **Security**:
   * Ensuring the security of user data was paramount. This guided our decisions on authentication methods, password hashing, and session management.
3. **User Experience**:
   * The end-user experience was always at the forefront of our decision-making process. We focused on creating an application that was easy to use, responsive, and visually appealing.
4. **Maintainability**:
   * We aimed to create a codebase that was clean, well-organized, and easy to maintain. This influenced our choice of technologies, as well as our approach to structuring the application.

**Lessons Learned**

1. **Flexibility in Technology Choices**:
   * **Lesson**: The initial decision to use PostgreSQL, followed by a switch to MongoDB, underscored the importance of flexibility in technology choices. It's crucial to remain open to changing the technology stack if it better suits the project’s evolving requirements.
   * **Takeaway**: Always prioritize the project’s needs over the desire to stick with a particular technology. Being adaptable can lead to more efficient and scalable solutions.
2. **Importance of Thorough Planning**:
   * **Lesson**: Proper planning and understanding of the project requirements were essential in making informed decisions, such as the choice of database and the structure of the user authentication system.
   * **Takeaway**: Spend adequate time in the planning phase to anticipate potential challenges and choose technologies that align with the project’s goals. This can prevent major setbacks during development.
3. **Security Best Practices**:
   * **Lesson**: Implementing a secure authentication system highlighted the importance of adhering to security best practices, such as using bcrypt for password hashing and managing sessions securely.
   * **Takeaway**: Never compromise on security. Even if it adds complexity to the development process, it is crucial for protecting user data and maintaining trust.
4. **User-Centric Design**:
   * **Lesson**: Designing with the end-user in mind was key to creating an intuitive and responsive application. User experience considerations influenced everything from UI design to error handling.
   * **Takeaway**: Always keep the user’s experience at the center of the design process. This approach ensures higher user satisfaction and engagement with the application.
5. **Effective Error Handling**:
   * **Lesson**: Implementing comprehensive error handling across the application made it easier to debug issues and provided users with clear feedback when things went wrong.
   * **Takeaway**: Effective error handling not only improves the development process but also enhances the user experience by providing clear guidance and preventing frustration.
6. **Collaboration and Communication**:
   * **Lesson**: The project’s success depended on effective collaboration and clear communication among team members. Whether discussing changes to the tech stack or addressing challenges, open communication ensured everyone was on the same page.
   * **Takeaway**: Foster a collaborative environment where team members feel comfortable sharing ideas and concerns. Regular communication is key to keeping the project on track.
7. **Iterative Development and Testing**:
   * **Lesson**: The iterative approach to development allowed for continuous testing and refinement of features. This helped in identifying issues early and making necessary adjustments without derailing the project.
   * **Takeaway**: Embrace an iterative development process with regular testing to catch and resolve issues early. This approach leads to a more stable and reliable final product.
8. **Handling Complexity with Modular Design**:
   * **Lesson**: Breaking down the application into smaller, modular components made it easier to manage complexity and maintain the codebase. Each component could be developed, tested, and updated independently.
   * **Takeaway**: Use a modular design approach to simplify development and maintenance. This allows for easier scaling and updating of the application in the future.

These lessons learned from the project will be invaluable in guiding future development efforts, ensuring that we continue to improve our processes and deliver high-quality applications.

**Did We Meet the Project Benchmarks?**

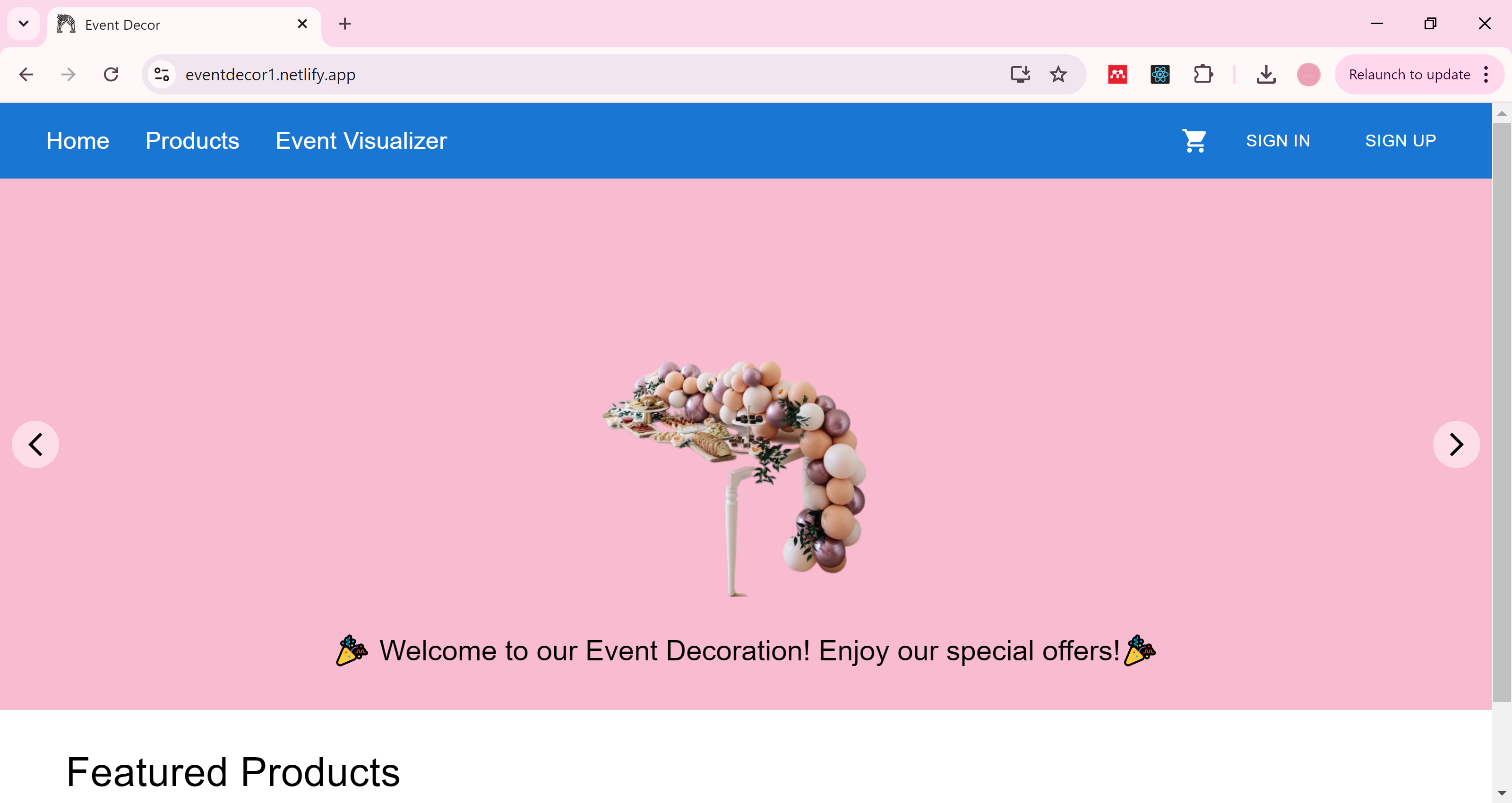
We think that we successfully met the benchmarks we set for ourselves at the beginning of the project. Here's a breakdown of how we achieved each of our goals:

1. **Implementation of User Authentication**:
   * **Benchmark**: Develop a secure user authentication system that supports both local and Google Firebase-based login.
   * **Outcome**: We implemented a robust authentication system using Passport.js, integrating bcrypt for secure password hashing and Google Firebase for social login. This system met our security and usability standards, allowing users to register and log in seamlessly.
2. **Database Management**:
   * **Benchmark**: Choose and integrate a database system that can handle the application's needs efficiently.
   * **Outcome**: Initially, we experimented with PostgreSQL but ultimately transitioned to MongoDB for its flexibility and ease of use with our project’s data structure. This change allowed us to manage the data more effectively and meet the application's requirements.
3. **Responsive UI/UX Design**:
   * **Benchmark**: Design a user-friendly interface that provides a smooth experience across different devices.
   * **Outcome**: The final product features a responsive and intuitive design, ensuring that users can easily navigate and interact with the application on both desktop and mobile devices. Feedback during testing was positive, indicating that the UI/UX goals were achieved.
4. **Error Handling and Validation**:
   * **Benchmark**: Implement comprehensive error handling and input validation to enhance user experience and system reliability.
   * **Outcome**: We incorporated extensive error handling and validation mechanisms, which improved the overall robustness of the application. Users received clear, helpful feedback on input errors, and the system maintained stability during unexpected conditions.
5. **Performance and Scalability**:
   * **Benchmark**: Ensure the application is optimized for performance and can scale with increased user activity.
   * **Outcome**: The application was optimized for performance through efficient database queries and minimal server-side processing. While scalability was tested within the project's scope, the modular design and use of MongoDB position the application for future growth.
6. **Security and Privacy**:
   * **Benchmark**: Maintain high standards of security, particularly in user authentication, data storage, and session management.
   * **Outcome**: Security measures, such as password hashing, secure session management, and data encryption, were rigorously implemented. This helped protect user data and maintained the privacy and integrity of the application.
7. **Meeting Deadlines**:
   * **Benchmark**: Complete each phase of development within the set deadlines.
   * **Outcome**: We adhered to the project timeline, delivering each milestone on schedule. This disciplined approach to time management ensured that we met all project benchmarks without compromising quality.

**User Guide**

**Screens of our system:**

**Home Page:** This is the main page of the website



Top section allows user to navigate different options:

**Home:** redirects to the main page of the website.

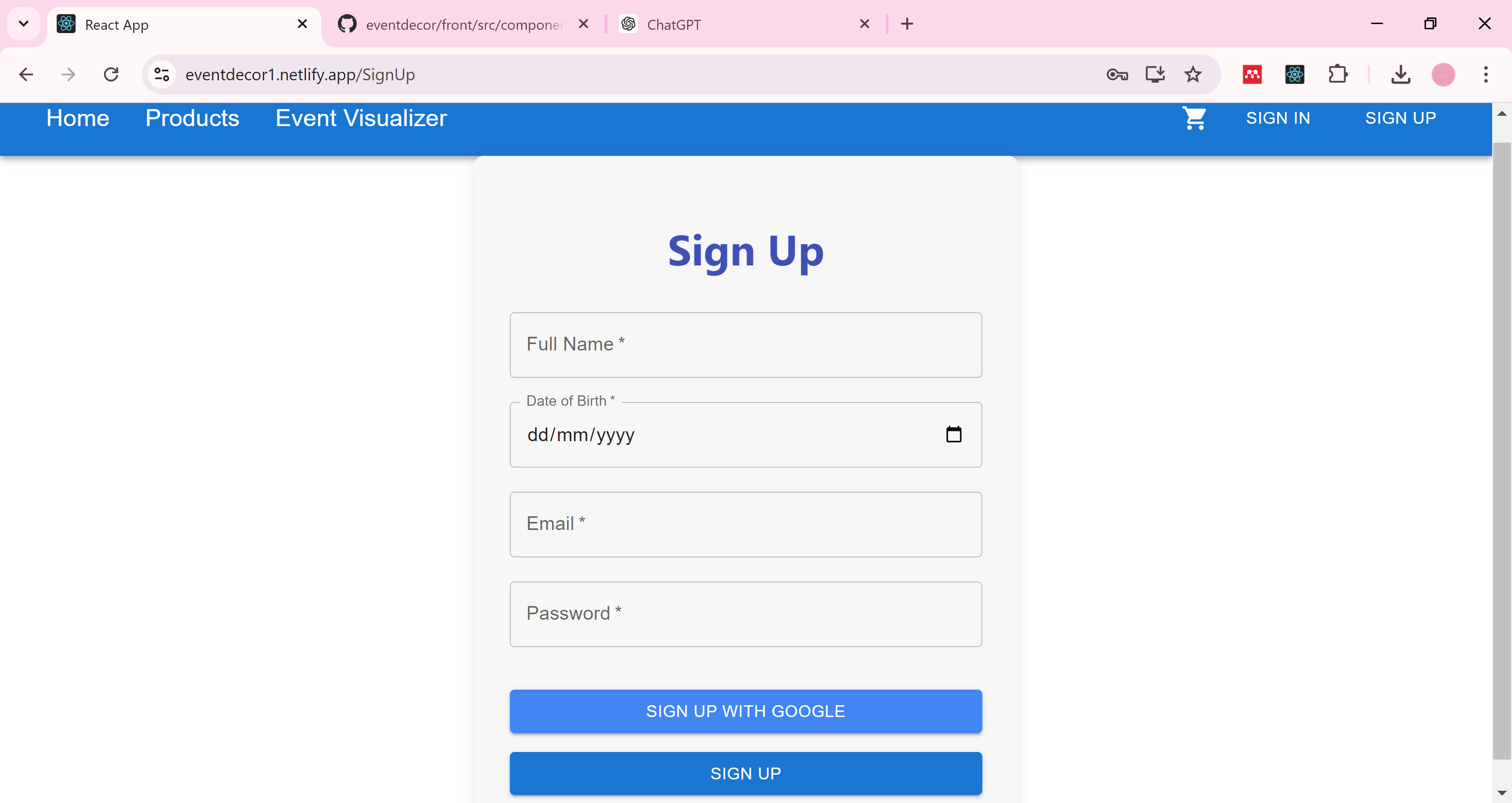
**Products :** take users to a page listing various products or services related to event decoration.

**Event Visualizer** **:** feature that allows users to visualize or customize event decorations.

**Sign In and Sign Up Buttons :** these buttons are for user authentication. "Sign In" is for existing users to log in, and "Sign Up" is for new users to create an account.

**Cart Icon :** leads to a shopping cart, where users can view or manage the items they intend to purchase.

**Sign Up forum page:** for new users to create an account.



**Full Name:** A required input field where the user enters their full name.

**Date of Birth:** A required date input field with a calendar icon. Users must select or enter their date of birth in the format dd/mm/yyyy.

**Email:** A required input field where the user enters their email address.

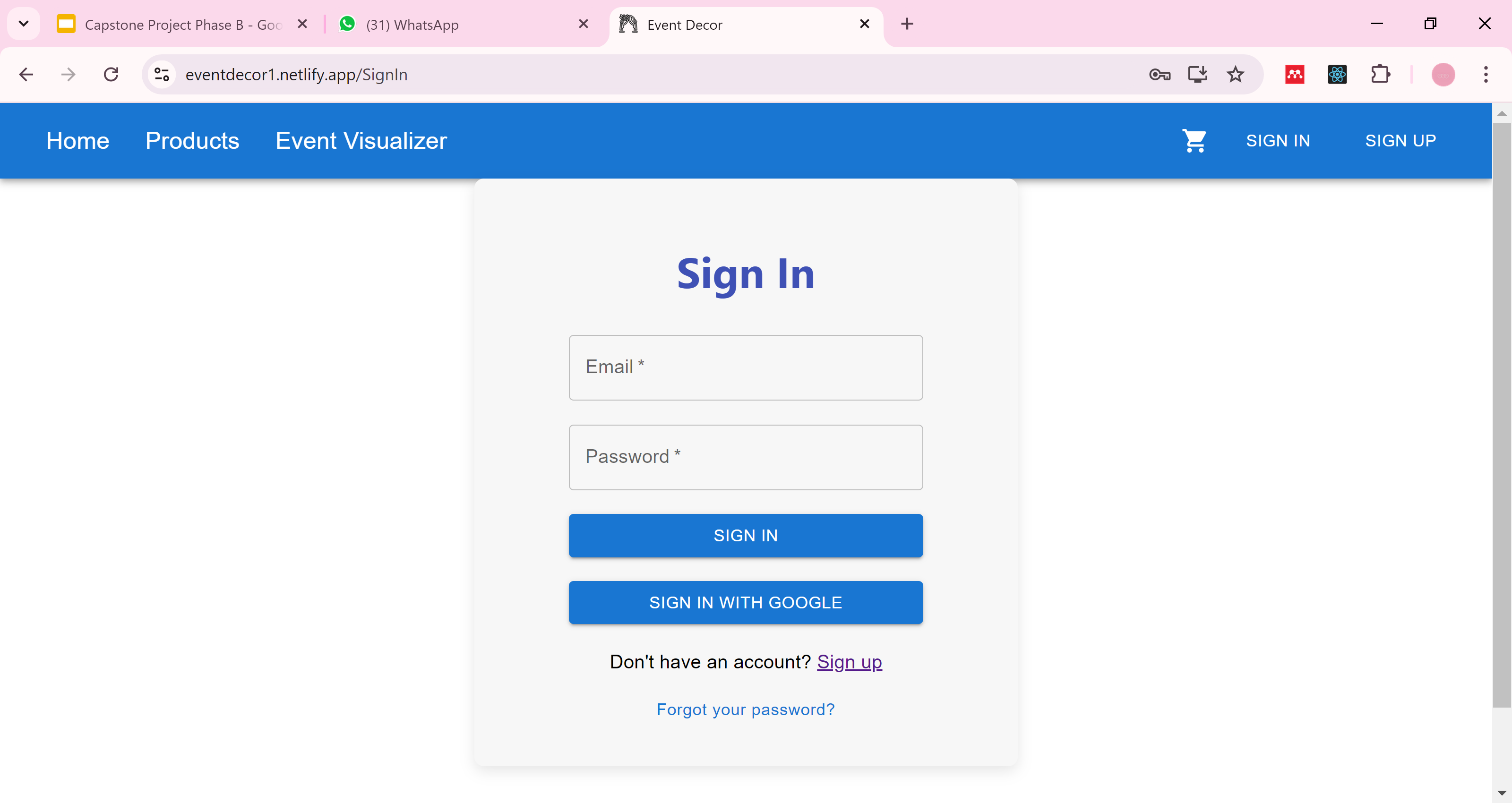
**Password:** A required input field where the user sets a password for their account.

* Sign Up Options

**Sign Up with Google:** This button that allows users to sign up using their Google account, simplifying the process by connecting with existing credentials.

**Sign Up Button:** This button submits the form to create the new user account with the provided information.

**Sign In Forum Page :** for existing users to log in.



**Email:** A required input field where the user enters their registered email address.

**Password:** A required input field where the user enters their password to access the account.

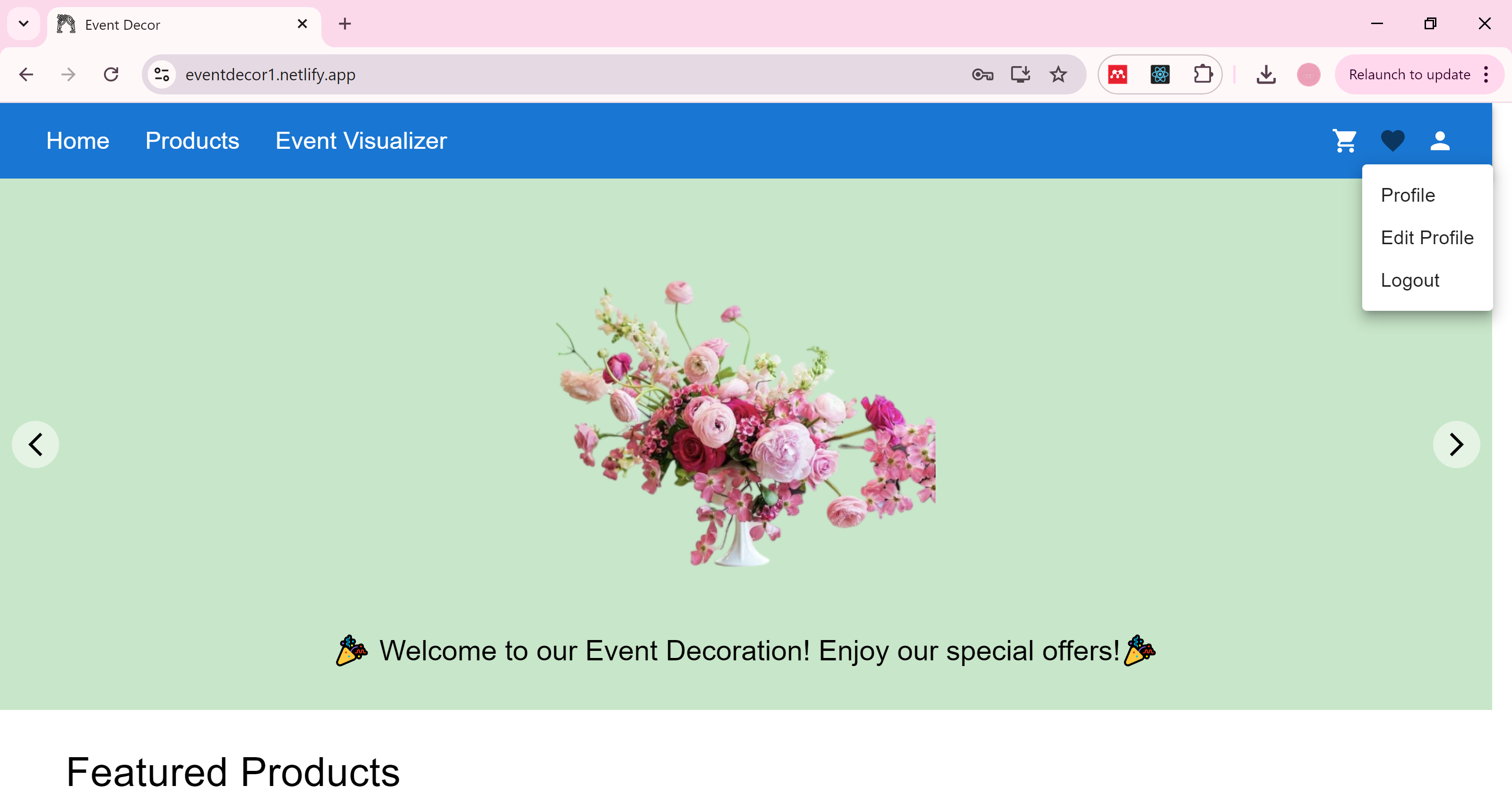
**Sign In Button:** This blue button submits the form, allowing the user to log in with the provided email and password.

**Sign In with Google:** This button provides an alternative method to sign in using a Google account. It’s a convenient option for users who prefer not to create a new password or for those who have registered with Google.

**Sign Up Link**: Below the form, there's a message that says, "Don't have an account? Sign up." This provides a link to the sign-up page for users who don't have an account yet.

**Forgot your Password:** Below the form,” forgot your password”, if the user forgot his password the system sends a code to the user mail and able him to change the password.

**Home page after User logIn:**



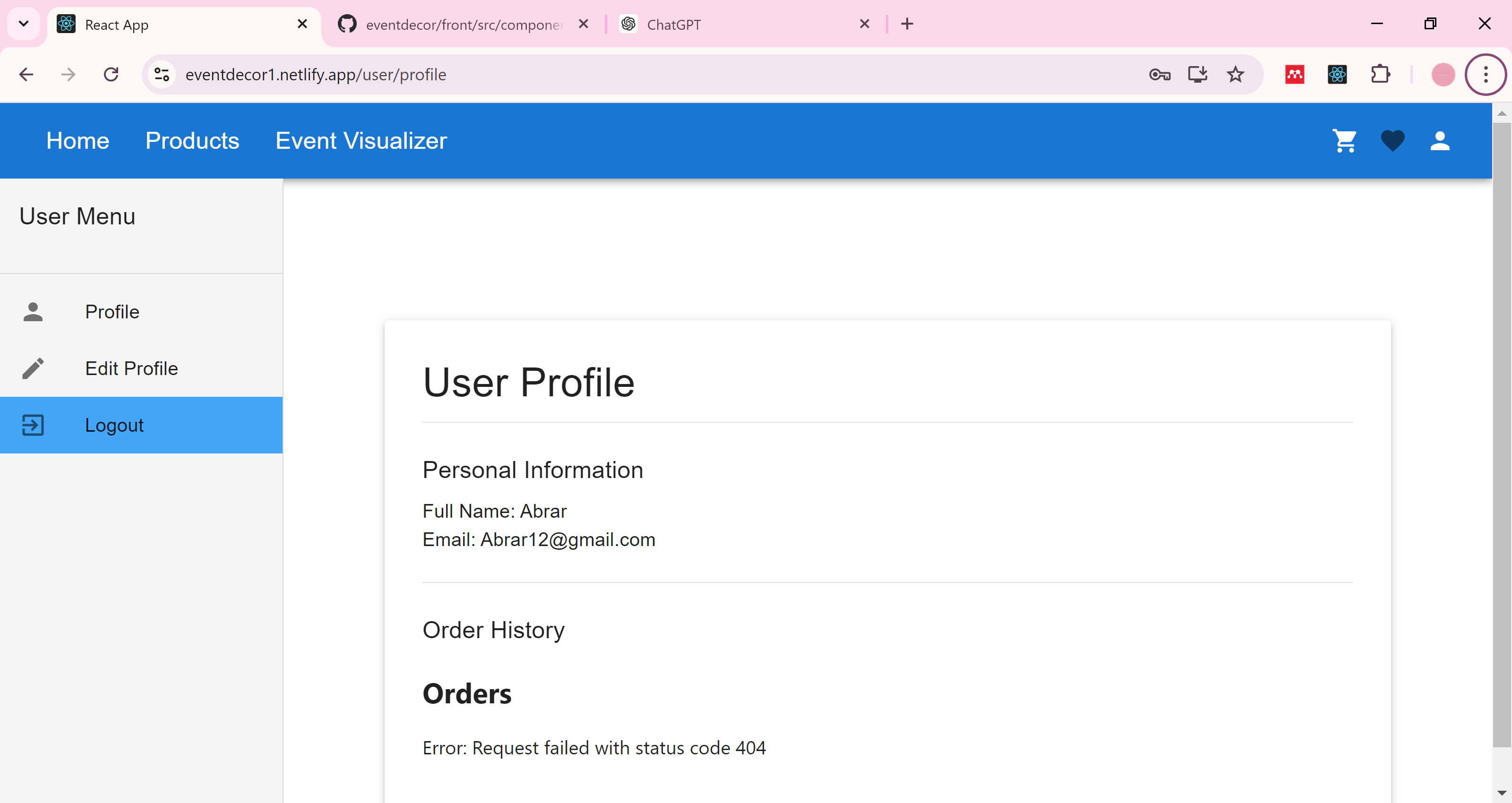
Additional options:

Heart Icon: allows View the favorite products that the user like.

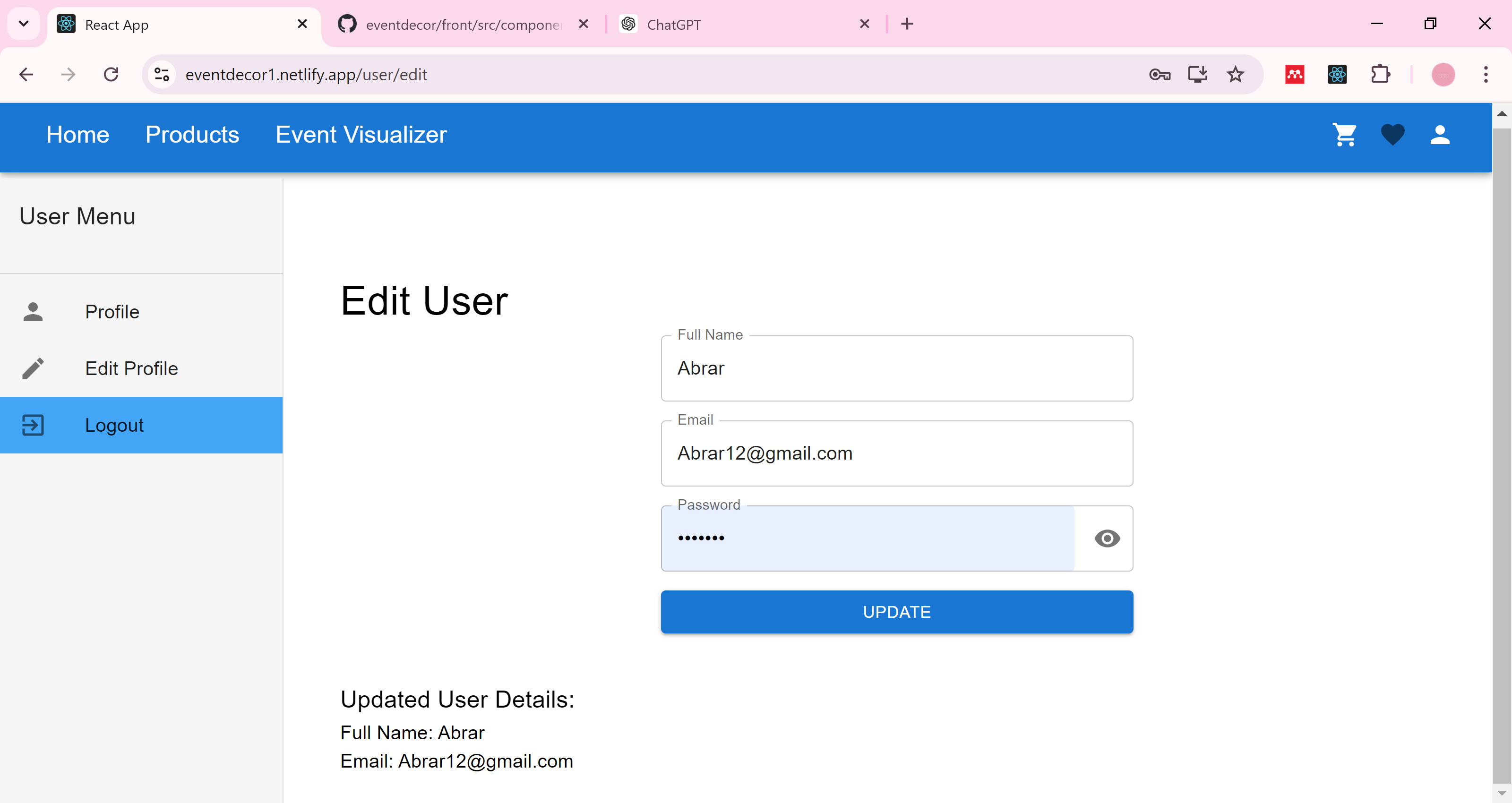
Person Icon options:User Menu

* + 1. Profile: allows user to view his profile.
    2. Edit Profile: allows user to edit his profile.
    3. Logout: allows user to logout from the webite.

**User Profile Page:**



**Edit Profile Page:**



**Products Page:**



**Filters Section :**

The filters section allows users to refine their search for products based on various criteria.

**Search:** A search bar where users can type keywords to find specific products.

**Category Dropdown:** Allows users to select a category of products, such as balloons, tableware, or lighting.

**Theme Dropdown:** Enables users to choose a theme, like birthday, wedding, or holiday decorations.

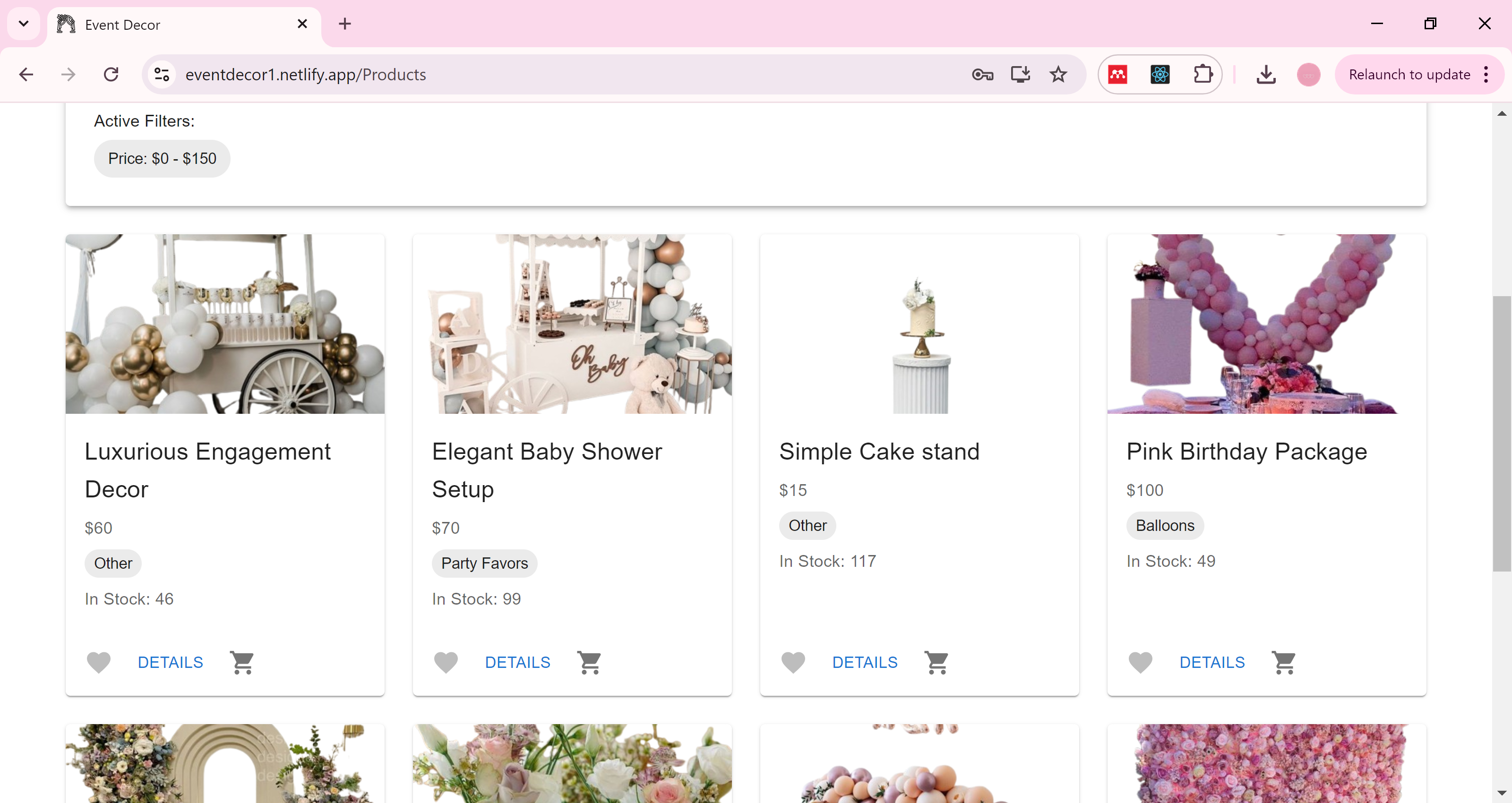
**Occasion Dropdown:** Lets users filter products based on the occasion, such as birthdays, weddings, or corporate events.

**Price Range Slider:** A slider that lets users adjust the price range for the products they want to see. In this screenshot, the price range is set from $0 to $150.

**Active Filters:** Displays the currently applied filters, which in this case is the price range.

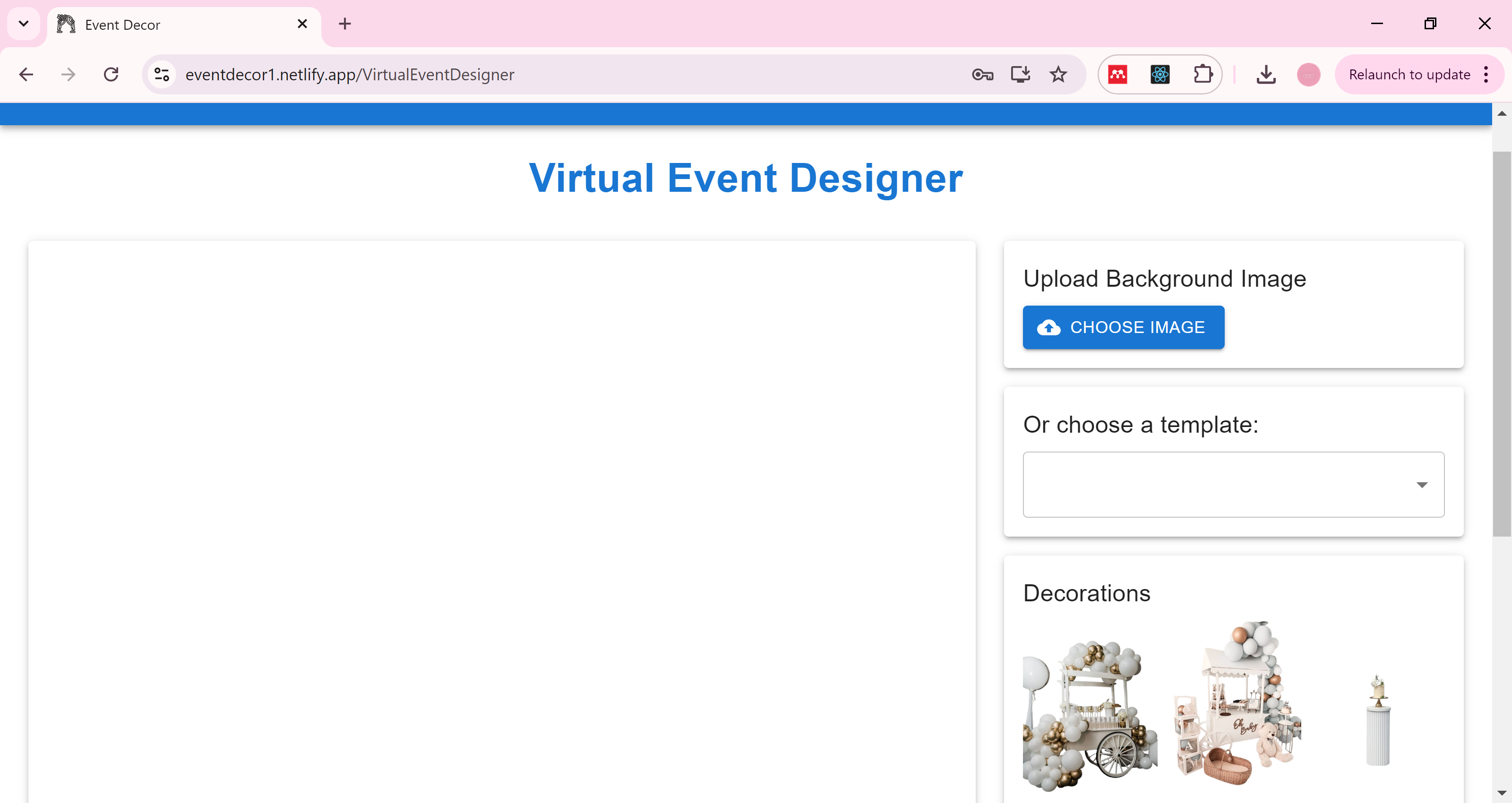
**Hide Filters:** Allows users to hide the filters section to get a broader view of the products.

**Clear Filters:** A button to clear all applied filters and reset the product display to its default state.



* The user can view each product details and add it to cart or favorite list.

**Event Visualizer Page:** help users visualize their event decorations before making a purchase or finalizing their plans.

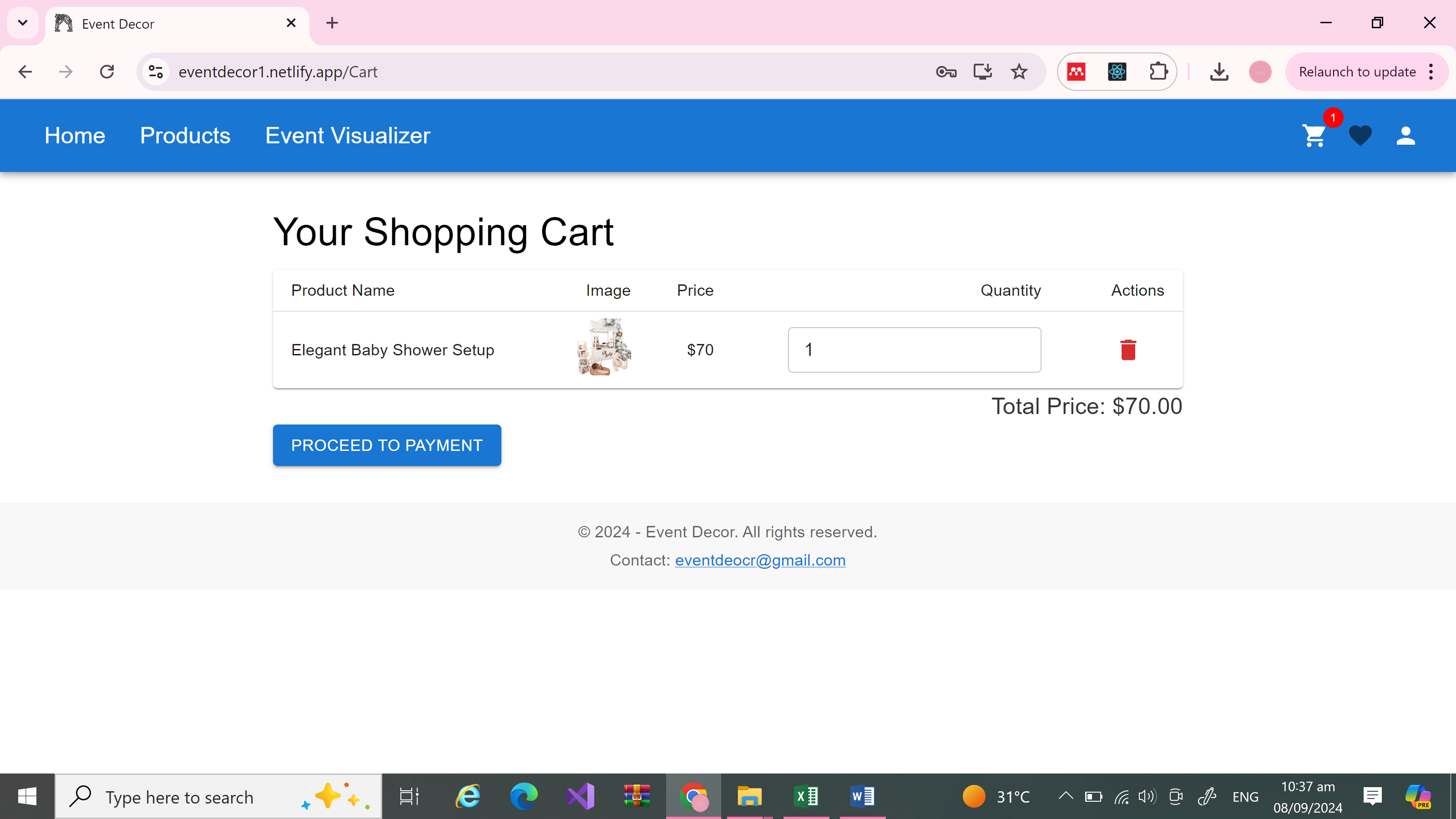


**Upload Background Image:** This section allows users to upload an image to serve as the background for their virtual event design. The button labeled "CHOOSE IMAGE" lets users select an image from their device to upload.

**Choose a Template:** Below the upload section, there is a dropdown menu labeled "Or choose a template:". Users can select a predefined template from this menu to use as a starting point for their design.

**Decorations Section:** This area displays a selection of decorations that users can add to their event design.

**Shopping Cart Page:** shows the user orders

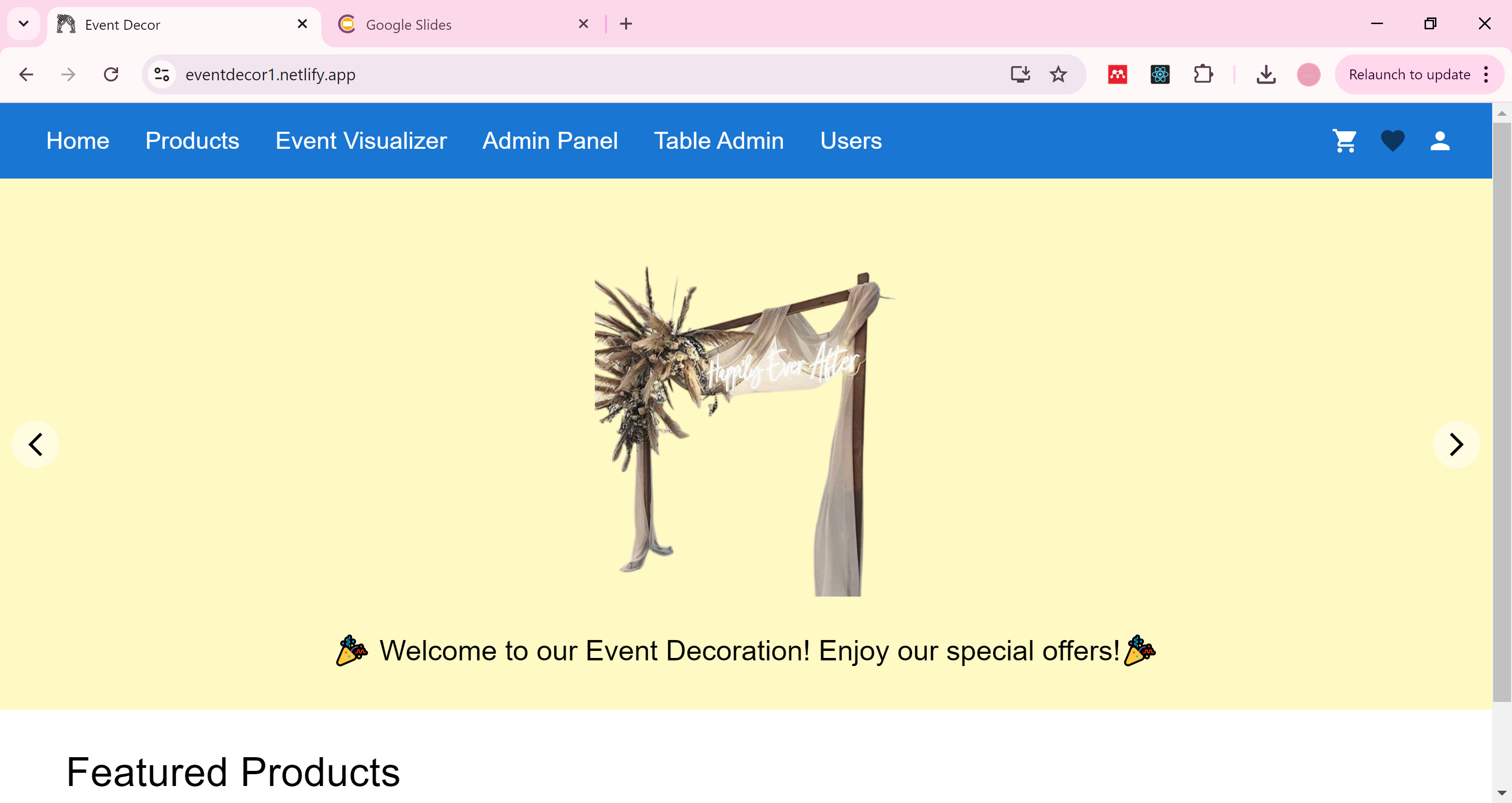


**Trash icon:** allowing users to remove it from the cart if they choose to do so.

**Quantity:** users can typically adjust this quantity using the input field provided.

**PROCEED TO PAYMENT**: button guiding the user to the next step in the checkout process, which is the payment.

**Home Page:** After Login as Admin



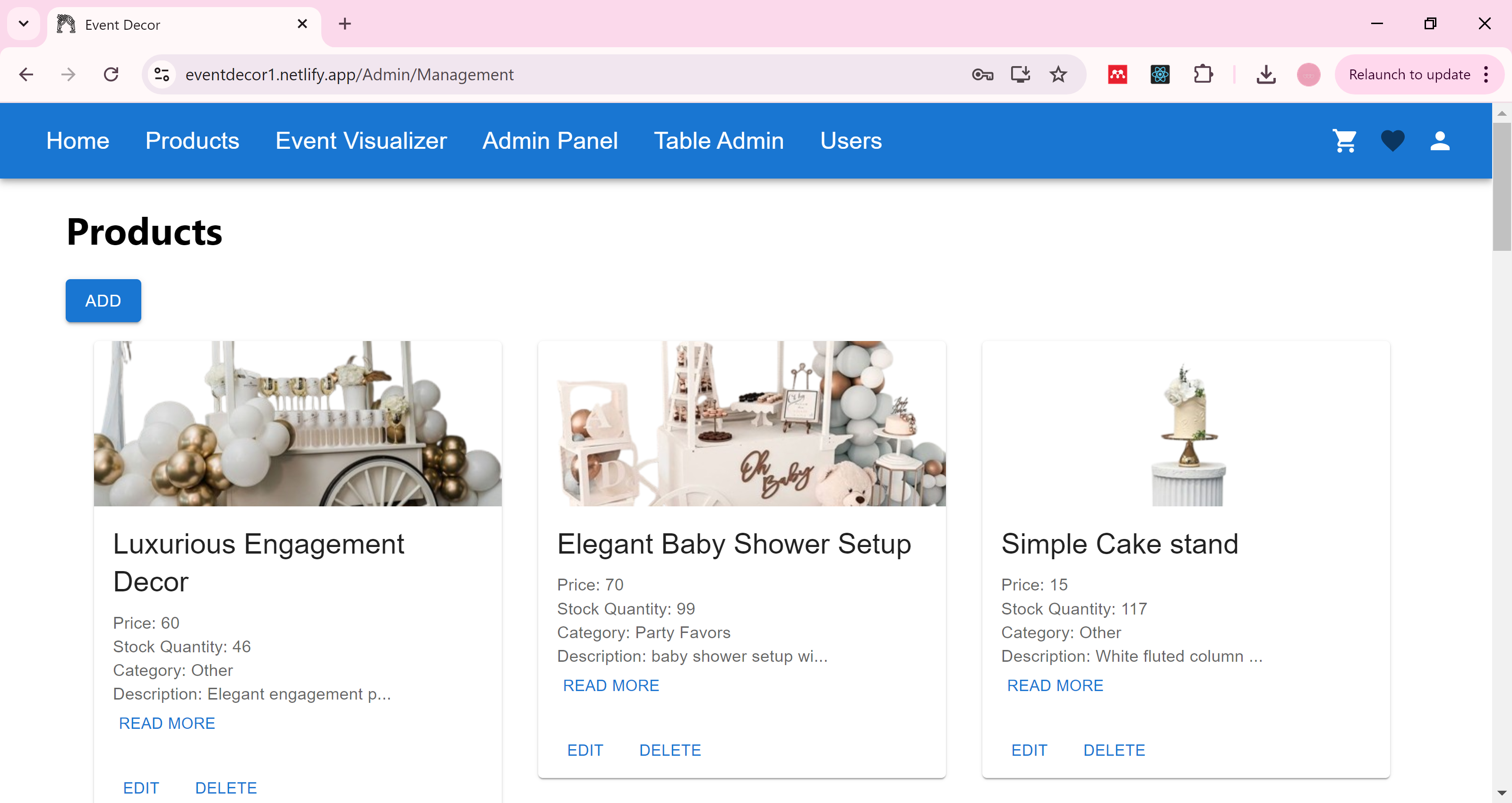
**Additional Options (if user was an Admin):**

**Admin Panel:** Allows admin to manage the products.

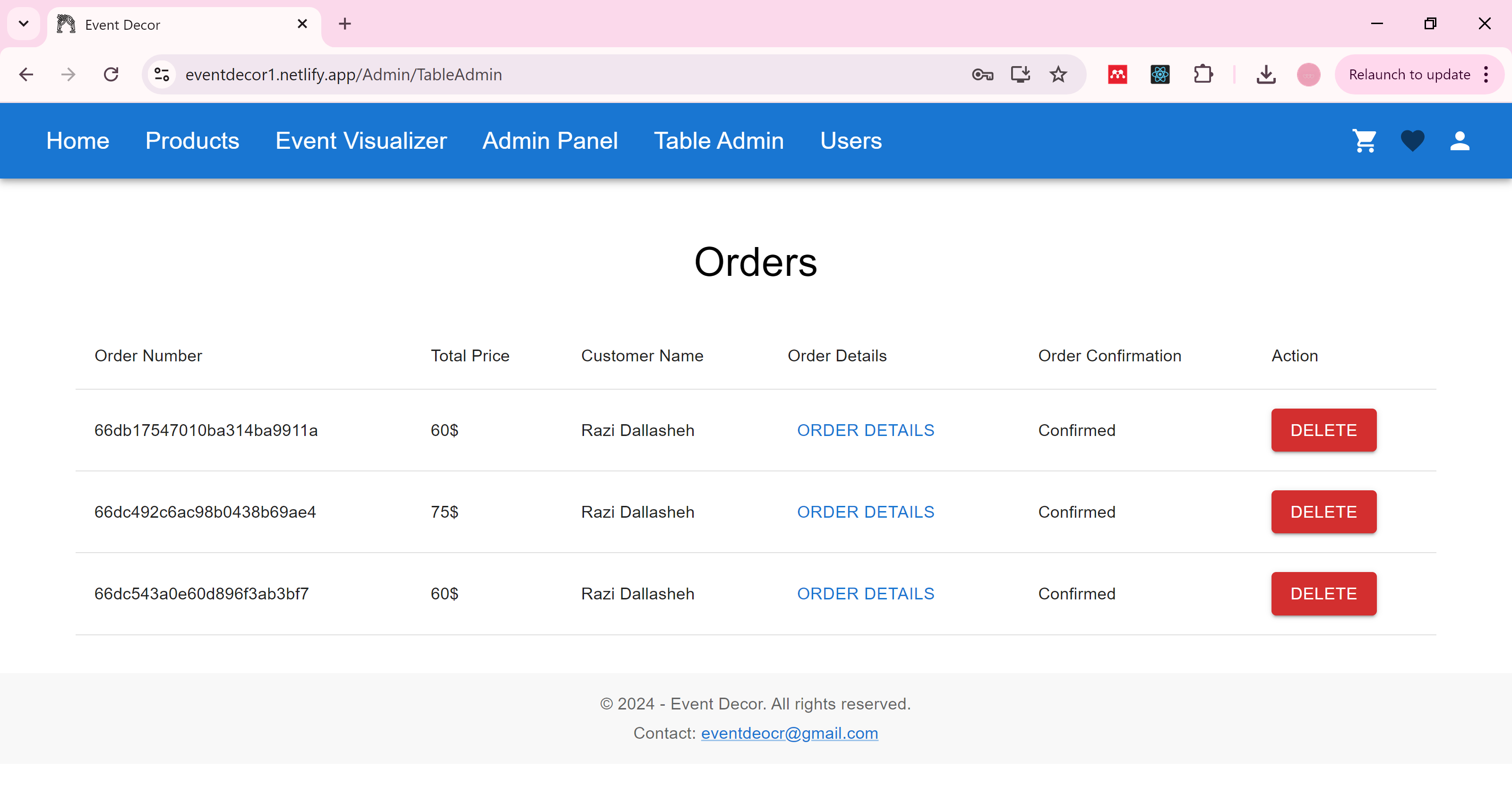
**Table Admin:** Allows admin to manage the orders.

**Users:** Allows admin to manage the users.

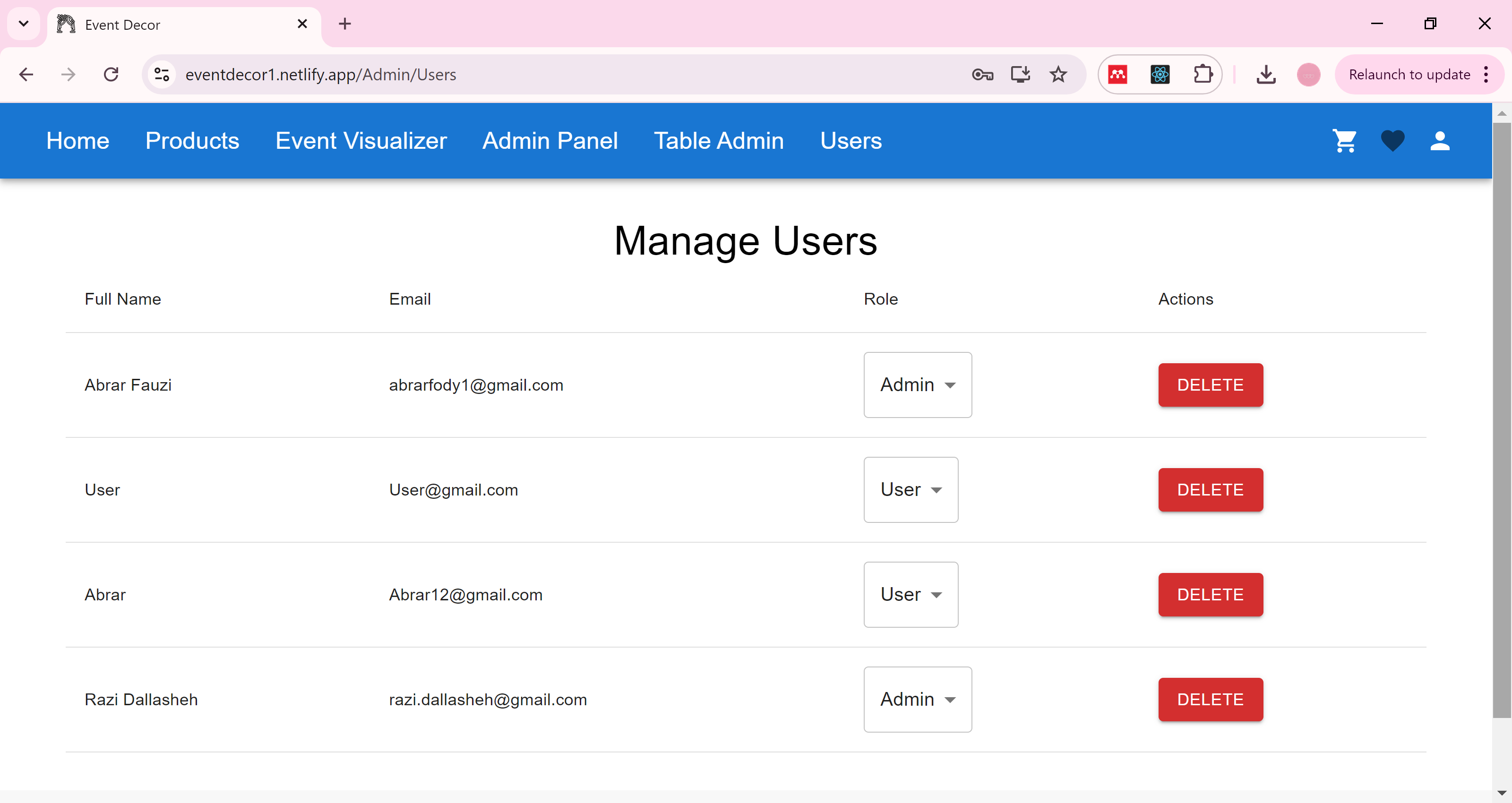
**Admin Panel Page:** allows admin to add new products, Edit and delete an existing product.



**Table Admin Page:** view the orders details, allows admin to delete order.



**Users page:** view the users details, allows admin to delete user and change roles.



**Maintenance Guide**

**The website URL here :** [**eventdecor1.netlify.app**](eventdecor1.netlify.app)

**Front-End:**

**Components:**

**Route:**

**AdminRoute.jsx:** returns a set of routes wrapped in a div element. Each route corresponds to a different part of the admin interface.

**UserRoute.jsx :** component provides a well-structured user profile section with a sidebar navigation and routing capabilities.

**Card:**

**ProductsCard.jsx:** displays product details and provides actions for editing and deleting the product.

**Footer:**

**Footer.jsx:** It displays the footer :current year, company name, and a contact email.

**Header:**

**Header.jsx**: It displays the header of the website

**Pages:**

**VirtualEventDesigner.jsx:** uploading and selecting backgrounds, adding decorations, resizing, dragging, and deleting elements

**Admin/management**

**Add**

**AddSingleProjact.jsx:** adding a single product.

**Edit:**Edit products.

**Management.jsx:** managing products.

**TableAdmin.jsx :** managment the orders.

**Cart :**

**Cart.jsx :** displaying items in a shopping cart, handling item deletion, updating quantities, and processing payments.

**Home:**

**Home.jsx:**display the home page.

**Products:**

**Products.jsx:** functionality of the product :filtering ,pagination to dialogs for product details and images.

**SignIn:**

**SignIn.jsx:** covers both email/password login and Google sign-in.

**SignUp:**

**SignUp.jsx:** handles both standard and Google sign-ups

**User:**

**Edit.jsx:** edit user profile.

**Profile.jsx**:displays the user profile.

**TableOrder.jsx:**displays the orders.

**Redux:** managing favorites,shopping cart,favorite and user with Redux and persisting it to cookies.

**reportWebVitals.js**: measure and report key web performance metrics

**Back-End:**

**Data**

**AddDecorations.js:** add a set of party decoration items to a MongoDB database using Mongoose. (don’t run this one because its causes problems within the database and also overwrites the existing database)

**Database Connection**: The script connects to the MongoDB database using Mongoose. The connection string is passed to mongoose. Connect.

**Decoration Data**: The array decorations contain multiple objects, each representing a party decoration with various attributes like name, price, description, stockQuantity, and more.

**Adding Decorations**: The addDecorations function loops through each decoration in the decorations array, creates a new PartyDecoration model instance, and saves it to the database.

**Error Handling**: If there is an error during the process, it will be logged in the console.

**Disconnect**: After all operations are complete, the database connection is closed using mongoose disconnect.

**Models**

**Order.js**: model for an “Order” collection in a MongoDB database.

**userId**: References the User model. This represents the user who placed the order.

**products**: An array of objects, each representing a product in the order. Each product object contains:

* **productId**: References the Product model.
* **quantity**: The quantity of the product ordered.
* **price**: The price of the product at the time of the order.

**totalAmount**: The total amount for the entire order.

**status**: The status of the order, which can be "Pending", "Completed", or "Canceled". By default, it's set to "Pending".

**orderDate**: The date the order was placed. It defaults to the current date and time.

**address**: An embedded document that includes:

* **street**
* **city**
* **postalCode**
* **country**

# **PartyDecoration.js :**

Mongoose schema and model for a PartyDecoration collection in a MongoDB database.

**name**:

* **Type**: String
* **Required**: true
* **Trim**: true (removes any leading or trailing whitespace)

**price**:

* **Type**: Number
* **Required**: true
* **Min**: 0 (ensures the price cannot be negative)

**description**:

* **Type**: String
* **Required**: true

**stockQuantity**:

* **Type**: Number
* **Required**: true
* **Min**: 0 (ensures the stock quantity cannot be negative)

**imageUrl**:

* **Type**: String
* **Required**: true

**category**:

* **Type**: String
* **Required**: true
* **Enum**: An array of valid categories: "Balloons", "Banners", "Tableware", "Lighting", "Party Favors", and "Other".

**color**:

* **Type**: String
* **Required**: true

**theme**:

* **Type**: String
* **Optional** (not required)

**occasion**:

* **Type**: String
* **Required**: true

**material**:

* **Type**: String
* **Optional** (not required)

**dimensions**:

* **Type**: Object containing length, width, and height, all of which are Number types. These are optional fields.

**isReusable**:

* **Type**: Boolean
* **Default**: false (if not specified, defaults to false)

**ageGroup**:

* **Type**: String
* **Enum**: "Kids", "Teens", "Adults", "All Ages". This restricts the field to one of these values.

**brand**:

* **Type**: String
* **Optional** (not required)

**weight**:

* **Type**: Number
* **Optional** (not required)

**packageQuantity**:

* **Type**: Number
* **Optional** (not required)

**User.js:** Mongoose schema and model for a User collection in a MongoDB database.

**googleId**:

* **Type**: String
* **Sparse**: true (allows multiple null or undefined values for the googleId)
* **Unique**: true (ensures that no two users can have the same googleId)
* This field is used to store the Google ID if the user signs up using Google authentication.

**fullName**:

* **Type**: String
* **Required**: true
* **Trim**: true (removes leading and trailing whitespace)
* Stores the user's full name.

**email**:

* **Type**: String
* **Required**: true
* **Unique**: true (ensures no two users can have the same email address)
* **Trim**: true
* Stores the user's email address.

**password**:

* **Type**: String
* **Required**: If the user is signing up using local authentication (authType is "local"), the password field is required.
* This field will store the user's password (usually hashed before storing).

**authType**:

* **Type**: String
* **Enum**: ["local", "google"] (restricts the value to either "local" or "google")
* **Required**: true
* Specifies whether the user signed up with local authentication (username and password) or Google.

**role**:

* **Type**: String
* **Enum**: ["User", "Admin"] (restricts the value to either "User" or "Admin")
* **Default**: "User"
* Specifies the role of the user, defaulting to a regular user role.

**address**:

* An embedded object storing the user's address with fields:
  + **street**: String
  + **city**: String
  + **postalCode**: String
  + **country**: String
* None of these fields are required, allowing for optional address information.

**Routes**

**Auth.js:** Express router module provides several routes to manage user authentication and account operations, including registration, login, Google login, updating user information, and retrieving user data.

**1. User Registration (/register):**

* **Validation**: Ensures all required fields (fullName, email, password) are provided.
* **User Existence Check**: Verifies that the user doesn't already exist by checking the email.
* **Password Encryption**: Uses bcryptjs to hash the user's password before saving.
* **User Creation**: Saves the new user with the local authentication method (authType: "local").
* **Email Confirmation**: Sends a confirmation email using nodemailer after successful registration.

**2. User Login (/login):**

* **Validation**: Ensures the user with the provided email exists.
* **Password Check**: Compares the provided password with the stored hashed password using bcryptjs.
* **Response**: Returns the user data if authentication is successful.

**3. Google Login (/google-login):**

* **User Existence Check**: Checks if the user already exists based on the email.
* **User Creation**: If the user doesn’t exist, it creates a new user with Google authentication (authType: "google").
* **Update Existing User**: If the user exists but doesn’t have a Google ID, it updates the user record to include the Google ID and switches the authType to google.

**4. Get User by ID (/user/:id):**

* **Validation**: Finds the user by ID.
* **Response**: Returns the user data if the user exists.

**5. Update User (/update/:id):**

* **Password Handling**: If a new password is provided, it is hashed before updating the user.
* **User Update**: Finds the user by ID and updates the provided fields.

**6. Get All Users (/users/all):**

* **Response**: Returns a list of all users in the database.

**7. Get User ID by Email (/user/id/:email):**

* **Validation**: Finds the user by email.
* **Response**: Returns the user ID if the user exists.

**Email Sending Configuration:**

* The email configuration uses nodemailer with Gmail as the service. Ensure the Gmail credentials (user and pass) are secure and ideally stored in environment variables instead of hardcoding them in the code.

**Security Considerations:**

* **Password Storage**: Passwords are securely hashed using bcryptjs before storing them in the database.
* **Environment Variables**: Consider using environment variables for sensitive information like Gmail credentials and database connection strings.
* **Error Handling**: Errors are caught and logged, and appropriate status codes are sent to the client.

**Cart.js**: Express router module provides several routes for managing orders within your application, including adding new orders, retrieving all orders, and fetching orders for a specific user.

**1. Add New Order (/add):**

* **Request Handling**: Accepts a new order through the request body.
* **Order Creation**: Creates a new order using the provided data and saves it to the database.
* **Response**: Sends a success message along with the newly created order.

**2. Get All Orders (/all):**

* **Fetch All Orders**: Retrieves all orders from the database.
* **Response**: Sends a list of all orders.

**3. Get All Orders with User and Product Details (/allOrders):**

* **Population**:
  + **User Details**: Populates the userId field to include user details in the order data.
  + **Product Details**: Populates the products.productId field to include the name and price of each product in the order.
* **Response**: Sends a list of all orders with populated user and product details.

**4. Get Order by User Email (/user/:email):**

* **Find Order by Email**: Retrieves a specific order based on the user's email address. (Note: The implementation seems intended to find orders by user email, but there appears to be a mistake in the variable name. It should be corrected.)
* **Response**: Sends the found order if it exists.

**5. Get Orders for Specific User (/user/:id/getOrders):**

* **Find Orders by User ID**: Retrieves all orders associated with a specific user ID.
* **Population**: Populates the products.productId field to include the name, price, and quantity of each product in the order.
* **Response**: Sends a list of orders for the specified user.

**Pay.js:** Express router module manages order processing, including creating new orders and handling payment processing.

**1. Sending Order Confirmation Emails**

* **Function**: sendOrderConfirmationEmail
* **Purpose**: Sends a confirmation email to the user after an order is created. It uses the nodemailer package to send emails via Gmail.
* **Usage**: Called internally after an order is successfully created.

**2. Creating a New Order (/user/:id/new\_order)**

* **Endpoint**: POST /user/:id/new\_order
* **Purpose**: Handles the creation of a new order for a specific user.
* **Key Steps**:
  + **Validate User**: Ensures the user ID exists in the database.
  + **Product Validation**: Checks that each product in the order exists and that there is sufficient stock.
  + **Create Order**: Saves the order to the database.
  + **Update Stock**: Decreases the stock quantity of each product by the quantity ordered.
  + **Send Email**: Sends an order confirmation email to the user.

**3. Processing Payment and Finalizing Orders (/process-payment)**

* **Endpoint**: POST /process-payment
* **Purpose**: Handles payment processing and finalizes the order.
* **Key Steps**:
  + **Validate Input**: Ensures all necessary data (user ID, cart data, total price, payment method, email, and address) are provided.
  + **Address Validation**: Confirms that the address contains all required fields (street, city, postal code, and country).
  + **User & Product Validation**: Validates the user and product information, similar to the new order route.
  + **Create Order**: Creates and saves the order with the payment information.
  + **Update Stock**: Adjusts the stock quantities for each product in the order.
  + **Send Email**: Sends a confirmation email to the user after successful payment and order creation.

**Products.js:**

**1. Add a New Party Decoration**

* **Endpoint**: POST /add
* **Functionality**:
  + This route handles adding a new party decoration to the database.
  + It validates the data before saving. If validation fails, it returns a 400 status with an error message.
  + On successful addition, it returns a 201 status with a success message and the added decoration object.
  + If there's a server-side issue, it returns a 500 status with an error message.

**2. Get All Party Decorations**

* **Endpoint**: GET /all
* **Functionality**:
  + This route retrieves all party decorations from the database.
  + Returns a 200 status with all decorations.
  + If there's a server-side issue, it returns a 500 status with an error message.

**3. Get Party Decoration Categories**

* **Endpoint**: GET /categories
* **Functionality**:
  + This route retrieves all distinct categories of party decorations.
  + Returns a 200 status with the list of categories.
  + If there's a server-side issue, it returns a 500 status with an error message.

**4. Get Party Decoration Themes**

* **Endpoint**: GET /themes
* **Functionality**:
  + This route retrieves all distinct themes of party decorations.
  + Returns a 200 status with the list of themes.
  + If there's a server-side issue, it returns a 500 status with an error message.

**5. Get Party Decoration Occasions**

* **Endpoint**: GET /occasions
* **Functionality**:
  + This route retrieves all distinct occasions associated with party decorations.
  + Returns a 200 status with the list of occasions.
  + If there's a server-side issue, it returns a 500 status with an error message.

**6. Get a Specific Party Decoration by ID**

* **Endpoint**: GET /get/:id
* **Functionality**:
  + This route retrieves a specific party decoration by its ID.
  + Returns a 200 status with the decoration if found.
  + If the decoration is not found, it returns a 404 status with an error message.
  + If the ID is invalid, it returns a 400 status with an error message.
  + If there's a server-side issue, it returns a 500 status with an error message.

**7. Update a Party Decoration**

* **Endpoint**: PUT /update/:id
* **Functionality**:
  + This route updates an existing party decoration by its ID.
  + Returns a 200 status with the updated decoration if successful.
  + If the decoration is not found, it returns a 404 status with an error message.
  + If there's a server-side issue, it returns a 500 status with an error message.

**8. Delete a Party Decoration**

* **Endpoint**: DELETE /delete/:id
* **Functionality**:
  + This route deletes a party decoration by its ID.
  + Returns a 200 status with a success message if the decoration is deleted.
  + If the decoration is not found, it returns a 404 status with an error message.
  + If the ID is invalid, it returns a 400 status with an error message.
  + If there's a server-side issue, it returns a 500 status with an error message.

**Order.js:** Express router module provides a route for deleting orders in your application.

**1. Delete an Order (/delete/):**

* Request Handling: Accepts a delete request for a specific order by using the order's unique ID passed as a route parameter.
* Order Deletion:
  + Retrieves the order by its ID and deletes it from the database using findByIdAndDelete().
  + If the order does not exist, returns a 404 error indicating that the order was not found.
* Response:
  + If the deletion is successful, the server sends a success message along with the deleted order details.
  + In the event of an error, such as an invalid order ID, a 400 (Bad Request) error is returned. Any other failures during the deletion process will result in a 500 (Internal Server Error) response.
* Logging: The order deletion request is logged for debugging purposes.

**Users.js:** Express router module provides routes for managing users within your application, including updating user roles, deleting users, and fetching all users.

**1. Update User Role (/update-role/):**

* Request Handling: Accepts a PUT request to update a user's role using the ID passed as a route parameter.
* User Search: Finds the user by their unique ID in the database.
* **Role Update:**
  + Updates the user's role using the role field provided in the request body.
  + Saves the updated user record back to the database.
* **Response:**
  + If the user is found and updated successfully, sends a success message along with the updated user data.
  + If the user is not found, responds with a 404 error.
  + If an error occurs during the process, responds with a 500 (Internal Server Error).

**2. Delete User (/delete/):**

* Request Handling: Accepts a DELETE request to remove a user by their unique ID passed as a route parameter.
* **User Deletion:**
  + Locates the user by their ID and deletes them from the database using findByIdAndDelete().
  + If the user is not found, returns a 404 error indicating that the user does **not exist.**
* **Response:**
  + If the deletion is successful, returns a success message.
  + If an error occurs during the deletion process, responds with a 500 error and an error message.

**3. Get All Users (/get):**

* Request Handling: Accepts a GET request to retrieve all users in the database.
* Fetch All Users: Retrieves all user records using find().
* Response:
  + If the users are successfully fetched, sends a list of all users with a 200 status.
  + If an error occurs, responds with a 500 error and logs the error to the server console for debugging.

**Db.js**

* It imports necessary modules: mongoose for MongoDB interaction and dotenv for loading environment variables.
* It configures dotenv to load environment variables from a .env file.
* The connectDB function is defined, which returns a promise. Inside this function:
* It uses mongoose.connect() to establish a connection to the MongoDB database using the URL and options specified in mongoOptions.
* If the connection is successful, it resolves the promise with the connection object (conn).
* If there's an error during the connection process, it rejects the promise with the error.

**Index.js:** includes the essentials for connecting to a database, handling routes, and enabling CORS for a specific origin.