Team 1:

Problem Statement: Build an E-Commerce Product Management App

You have **3 hours** to create a **Product Management Application** for an e-commerce platform. The app will allow users to view, add, edit, delete, and search for products. It will also demonstrate essential React concepts like state management, routing, and styling. Follow the detailed requirements below to complete the project.

Features & Requirements

1. Home Page (Product Listing)

- Fetch the product list dynamically from a **JSON Server** using the **useEffect** hook.
- Display the list of products in a tabular or card format, showing:
  - **Name** of the product.
  - **Price** (in a user-preferred currency).
  - **Description**.
- If no products exist in the database, display the message: *"No Products Available"* using **conditional rendering**.
- Add a "Delete" button to each product, allowing users to remove a product from the server.

2. Add Product Page

- Create a form with the following fields:
  - **Name** (text, required).
  - **Price** (number, required).
  - **Description** (text, optional).
- Use **controlled components** to manage form inputs with React **state**.
- Validate the form to ensure required fields are not empty.
- On form submission:
  - Send a POST request to the **JSON Server** to save the new product.
  - Redirect the user to the Home Page after successfully adding the product.

3. Edit Product Page

- Allow users to navigate to the **Edit Product Page** by clicking an "Edit" button on a product card.
- Use **react-router-dom** to navigate between routes and **useParams** to pass the product’s unique ID in the URL.
- Fetch the product details using the ID and prefill the form with its current data.
- Allow users to update product details using a PUT request to the **JSON Server**.
- Redirect the user back to the Home Page after successfully saving the updates.

4. Delete Product

- On the Home Page, include a "Delete" button for each product.
- When the delete button is clicked:
  - Prompt the user for confirmation using a simple window.confirm dialog.
  - Remove the product from the server using a DELETE request.

    - Update the product list dynamically to reflect the deletion.

5. Styling

- Use **styled-components** to create reusable, dynamic, and visually appealing styles for the application.
- Design the product list, form, and buttons with clear, modern aesthetics.
- Use conditional styling to highlight active elements (e.g., selected product, form focus).

6. Global State Management

- Use **React Context** to manage user preferences, such as:
    - Displaying prices in different currencies (e.g., USD, EUR, INR).
    - Saving the selected currency globally so it applies across the app.

7. Search Functionality

- Add a **search bar** at the top of the Home Page to filter products dynamically by:
    - **Name**.
    - **Category** (if applicable).
- Use React **state** to manage the search input and filter the product list in real-time as the user types.

8. Ref and forwardRef

- Use a **ref** to focus on the search bar automatically when the Home Page loads.
- Implement **forwardRef** to dynamically manage form inputs if required.

Implementation Notes

- **Routing**: Use **react-router-dom** to create navigation between the Home Page, Add Product Page, and Edit Product Page.
- **API**: Use **JSON Server** as a mock backend to manage product data. Run it locally with:
- **Deployment**: Deploy the application to **Vercel** and ensure all features are functional in the deployed version.

Deliverables

1. A working **React application** hosted on **Vercel**.

Team 2:

Problem Statement: Build a Library Management App

You have **3 hours** to create a **Library Management Application**. The app will help users manage a collection of books, including viewing, adding, editing, and deleting book entries. You'll use React's core features, including routing, state management, and styling, while integrating with a mock backend.

Features & Requirements

1. Home Page (Book Listing)

- Fetch a list of books from a **JSON Server** using the **useEffect** hook.
- Display each book in a table or card format, showing the following details:
  - **Title** of the book.
  - **Author**.
  - **Genre** (e.g., Fiction, Non-fiction, Mystery, etc.).
- If no books are available, display the message: *"No books available"* using **conditional rendering**.
- Add a "Delete" button for each book, allowing users to delete a book. Use an event handler to send a DELETE request to the server.

2. Add Book Page

- Create a form with the following fields:
  - **Title** (text, required).
  - **Author** (text, required).
  - **Genre** (dropdown or text, required).
- Use **controlled components** to manage the form inputs with React **state**.
- Validate the form to ensure that all fields are filled out.
- On form submission:
  - Send a POST request to the **JSON Server** to save the new book.
  - Redirect the user to the Home Page after successfully adding the book.

3. Edit Book Page

- Allow users to edit the details of a book:
  - Navigate to the **Edit Book Page** by clicking an "Edit" button on the book listing.
  - Use **react-router-dom** to navigate between pages and **useParams** to retrieve the book's ID from the URL.
  - Fetch the book's data from the server using the ID and prefill the form with its current details.
  - Allow users to update the book's information and send a PUT request to save changes.
  - Redirect the user back to the Home Page after updating the book successfully.

4. Search Functionality

- Add a **search bar** at the top of the Home Page to dynamically filter books by their title.
- Use **state** to manage the search input. As the user types, update the displayed book list to match the search query.

5. Styling

- Use **styled-components** to create dynamic, reusable styles for the app.
- Style the book list, forms, and buttons with modern and user-friendly designs.
- Add hover effects for book entries and buttons to enhance the user experience.

6. Navigation

- Use **react-router-dom** to handle navigation between:
  - **Home Page**: Displays the list of books.
  - **Add Book Page**: Allows users to add a new book.
  - **Edit Book Page**: Enables editing of existing books.
- Include a navigation bar or links for easy access to these pages.

7. Global Settings with React Context

- Add a **theme toggle** to switch between "Light" and "Dark" themes using **React Context**.
- Apply the selected theme globally to all components styled with **styled-components**.

8. Ref and forwardRef

- Use a **ref** to automatically focus on the search bar when the Home Page loads.
- Use **forwardRef** to manage reusable input components in the form (optional for advanced learners).

9. Data Storage

- Use **JSON Server** as a mock backend to manage book data.
  - Create a db.json file with sample book data.

10. Deployment

- Deploy your Library Management App to **Vercel**.

Deliverables

1. A **fully functional Library Management App** hosted on Vercel.

Team 3:

Problem Statement: Build a Pet Adoption App

You have **3 hours** to create a **Pet Adoption Application**. This app will help users manage and display pets available for adoption while learning essential React concepts and techniques. Follow the detailed requirements below.

Features & Requirements

1. Home Page (Pet List)

- Fetch a list of pets from a **JSON Server** (db.json) and display them in a card layout. Each card should show:
  - **Pet Name**
  - **Breed**
  - **Age**
  - **Adoption Status** (e.g., *Available* or *Adopted*).
- Include a "Delete" button on each card to remove the pet from the server.
- If there are no pets in the database, display the message: *"No pets available for adoption"* using **conditional rendering**.

2. Add Pet Page

- Create a form with the following fields:
  - **Name** (text, required)
  - **Breed** (text, required)
  - **Age** (number, required)
  - **Status** (dropdown with options: *Available* and *Adopted*, default to *Available*).
- Use **controlled components** to manage form input with React **state**.
- On submission:
  - Validate the form (ensure no fields are empty).
  - Send a POST request to the **JSON Server** to save the new pet.
  - Redirect the user back to the Home Page after successfully adding the pet.

3. Edit Pet Details

- Allow users to update pet details using a dedicated **Edit Page**.
- Use **react-router-dom** to navigate to the edit page. Pass the pet's unique ID in the URL using **useParams**.
- Fetch the pet's current details using the ID and prefill the form with its data.
- Update the pet's details on form submission by sending a PUT request to the **JSON Server**. Redirect back to the Home Page after updating.

4. Search Pets

- Add a **search bar** at the top of the Home Page to filter pets dynamically by:
  - **Name** (case-insensitive).
  - **Breed** (case-insensitive).
- Use **state** to manage the search input and filter the displayed pets based on the input.

5. Global State for Filters

- Use **React Context** to create a toggle filter that switches between showing:
  - **All Pets**.
  - Only *Available* pets.
  - Only *Adopted* pets.
- Display the filter toggle as radio buttons or a dropdown above the Pet List.

6. Styling with styled-components

- Use **styled-components** to style the app.
- Design visually appealing cards for pets, the search bar, and the forms.
- Add hover effects for buttons and cards.
- Apply conditional styling to indicate the adoption status (e.g., green for *Available*, gray for *Adopted*).

7. Data Storage

- Use a **JSON Server** to store and retrieve pet data.
  - Set up the server locally with db.json.

8. Deployment on Vercel

- Deploy the app on **Vercel** to make it accessible online.

Deliverables

By the end of this task, you should deliver:

1. A **fully functional Pet Adoption App** hosted on Vercel.

Team 4:

Problem Statement: Build a Space Explorer App

You have **3 hours** to create a **Space Explorer Application**. This app will help users explore, add, and manage information about celestial objects. You will implement features to learn React concepts like props, state, forms, routing, context, and dynamic styling. Follow the detailed requirements below.

Features & Requirements

1. Home Page (Celestial Object List)

- Fetch a list of celestial objects from a **JSON Server** (db.json) and display them in a card layout. Each card should show:
  - **Name** of the celestial object.
  - **Type** (e.g., Star, Planet, Nebula).
  - **Distance** from Earth (in light-years).
- Include a "Delete" button on each card to remove the object from the database.
- If the list is empty, display the message: *"No celestial objects to display"* using **conditional rendering**.

2. Add Object Page

- Create a form with the following fields:
  - **Name** (text, required).
  - **Type** (dropdown with options: Star, Planet, Nebula, required).
  - **Distance from Earth** (number, required).
- Use **controlled components** to manage form inputs with React **state**.
- On form submission:
  - Validate the form to ensure all fields are filled.
  - Send a POST request to the **JSON Server** to save the new object.
  - Redirect the user back to the Home Page after successfully adding the object.

3. Edit Object Details

- Allow users to update details of celestial objects using a dedicated **Edit Page**.
- Use **react-router-dom** to navigate to the edit page. Pass the object's unique ID in the URL using **useParams**.
- Fetch the object's current details using the ID and prefill the form with its data.
- Send a PUT request to the **JSON Server** to save the updated details.
- Redirect the user back to the Home Page after updating.

4. Global Filters with React Context

- Use **React Context** to create a global filter that allows users to:
  - Show **All Objects**.
  - Filter by **Type** (e.g., Stars only, Planets only, or Nebulas only).
- Place the filter as a dropdown or radio buttons above the object list. The selection should persist across navigation.

5. Search Celestial Objects

- Add a **search bar** at the top of the Home Page to dynamically filter objects by:
  - **Name** (case-insensitive).
  - **Type** (case-insensitive).
- Use **state** to manage the search input and filter results in real-time.

6. Styling with styled-components

- Use **styled-components** for styling the application.
- Create visually appealing cards for celestial objects with space-themed colors and animations.
- Add hover effects for buttons and cards.
- Use conditional styling to visually differentiate celestial object types (e.g., unique colors for Stars, Planets, and Nebulas).

7. Data Storage

- Use a **JSON Server** to manage celestial object data.
  - Set up the server locally with db.json.

8. Deployment on Vercel

- Deploy the app on **Vercel** to make it accessible online.

Deliverables

By the end of this task, you should deliver:

1. A **fully functional Space Explorer App** hosted on Vercel.

Team 5:

Problem Statement: Build a Sports Score Tracker App

You have **3 hours** to create a **Sports Score Tracker Application**. The app will allow users to view, add, edit, and search for live match scores. This project will use key React concepts, libraries, and techniques as outlined below.

Features & Requirements

1. Scoreboard (Home Page)

- Fetch the list of ongoing matches dynamically from a **JSON Server** using the **useEffect** hook.
- Display each match in a card format, showing:
    - **Team Names** (e.g., Team A vs. Team B).
    - **Current Score** for both teams.
    - **Match Status** (e.g., Ongoing, Completed).
- If no matches are available, display the message: *"No matches currently available"* using **conditional rendering**.
- Include a "Delete" button for each match, allowing users to remove a match from the server.

2. Add Match Page

- Create a form with the following fields:
    - **Team A Name** (text, required).
    - **Team B Name** (text, required).
    - **Current Score** for each team (number, required).
    - **Match Status** (dropdown: Ongoing/Completed, required).
- Use **controlled components** to manage the form inputs with React **state**.
- Validate the form to ensure all fields are filled out correctly.
- On form submission:
    - Send a POST request to the **JSON Server** to save the new match.
    - Redirect the user to the Scoreboard after successfully adding the match.

3. Edit Match Page

- Allow users to update match details, including the score and status:
    - Navigate to the **Edit Match Page** by clicking an "Edit" button on a match card.
    - Use **react-router-dom** to navigate between pages and **useParams** to retrieve the match's ID from the URL.
    - Fetch the match details from the server using the ID and prefill the form with its current data.
    - Allow users to update the match's information and send a PUT request to save changes.
    - Redirect the user back to the Scoreboard after successfully updating the match.

4. Search Matches

- Add a **search bar** at the top of the Scoreboard to dynamically filter matches by:
    - **Team Name** (e.g., Team A or Team B).

- Use **state** to manage the search input. As the user types, update the displayed match list to show only matches that include the search query.

5. Styling

- Use **styled-components** to create dynamic and reusable styles for the app.
- Style the match cards, forms, and buttons with a modern and user-friendly design.
- Add hover effects and animations to enhance user experience.

6. Navigation

- Use **react-router-dom** to handle navigation between:
    - **Scoreboard** (Home Page): Displays the list of matches.
    - **Add Match Page**: Allows users to add a new match.
    - **Edit Match Page**: Enables editing of existing matches.
- Include a navigation bar or links for seamless transitions between pages.

7. Data Storage

- Use **JSON Server** as a mock backend to manage match data.
    - Create a db.json file with sample match data.

8. Deployment

- Deploy your **Sports Score Tracker App** to **Vercel**.

Deliverables

1. A **fully functional Sports Score Tracker App** hosted on Vercel.

Team 6:

Problem Statement: Build a Cooking Timer App

You have **3 hours** to create a **Cooking Timer Application** that allows users to manage and track cooking timers for various recipes. The app will display timers, enable users to create and edit timers, and notify them when the timers are completed. Use React and related libraries to implement the following features.

Features & Requirements

1. Timer List (Home Page)

- Fetch a list of timers from a **JSON Server** using the **useEffect** hook.
- Display the list of timers in a user-friendly format, showing:
  - **Recipe Name** (e.g., Spaghetti, Cake, etc.).
  - **Time Remaining** for each timer.
- If no timers are available, display the message: *"No timers set"* using **conditional rendering**.
- Each timer should show how much time is left, which can be updated periodically (using **setInterval** or a similar approach).

2. Add Timer Page

- Create a form to add a new timer with the following fields:
  - **Recipe Name** (text, required).
  - **Duration** (number, in minutes, required).
- Use **controlled components** to manage form inputs with React **state**.
- On form submission:
  - Add the timer to the list (send a POST request to the **JSON Server**).
  - Redirect the user to the **Timer List** (Home Page) after adding the new timer.

3. Edit Timer Page

- Allow users to edit existing timers:
  - Use **react-router-dom** to navigate to an Edit page.
  - Use **useParams** to retrieve the timer ID from the URL.
  - Fetch the timer details from the server and prefill the form with its current data.
  - Allow users to update the recipe name and duration.
  - Send a PUT request to the **JSON Server** to save the updated timer.
  - Redirect the user back to the **Timer List** after updating the timer.

4. Global Settings (Sound Notification)

- Use **React Context** to manage global settings such as sound notifications.
- When a timer reaches 0, play a sound (use the HTML Audio API) to notify the user that the timer is complete.
- The user should be able to toggle sound notifications on or off globally, which will apply across the app.

5. Timer Functionality

- Each timer should:

    - Start counting down from the specified duration when created or edited.
    - Update the remaining time periodically and display it dynamically.
    - Automatically trigger the sound notification when the timer reaches 0.
    - Be deletable by the user (remove from the list with a DELETE request to the **JSON Server**).

6. Styling

- Use **styled-components** to create reusable styles for the app.
- Design a clean and user-friendly interface for the timer list, form, and buttons.
- Add hover effects, progress bars, or countdown visualizations to enhance the user experience.

7. Navigation

- Use **react-router-dom** to manage routing between:
    - **Timer List (Home Page)**: Displays the list of timers.
    - **Add Timer Page**: Allows users to create a new timer.
    - **Edit Timer Page**: Allows users to edit an existing timer.
- Include a navigation bar or links for easy transitions between pages.

8. Data Storage

- Use **JSON Server** as a mock backend to manage timer data.
    - Create a db.json file with sample timer data (e.g., recipe name and duration).

9. Deployment

- Deploy your **Cooking Timer App** to **Vercel**.

Deliverables

1. A **fully functional Cooking Timer App** hosted on **Vercel**.

Team 7:

Problem Statement: Build a Gardening Planner App

You have **3 hours** to build a **Gardening Planner Application** that helps users manage and track their plants, including details about watering schedules and plant types. This app will allow users to view, add, edit, and search for plants in their garden, and offer a simple global filter to categorize plants. The project will showcase key React concepts and libraries as outlined below.

Features & Requirements

1. Plant List (Home Page)

- Fetch a list of plants from a **JSON Server** using the **useEffect** hook.
- Display each plant's details, such as:
  - **Plant Name** (e.g., Rose, Aloe Vera).
  - **Plant Type** (e.g., Indoor, Outdoor).
  - **Watering Frequency** (e.g., Every 3 days).
- If no plants are available, display a message: *"No plants to display"* using **conditional rendering**.
- Each plant card should show basic information and allow users to edit or delete the plant.

2. Add Plant Page

- Create a form that includes the following fields:
  - **Plant Name** (text, required).
  - **Plant Type** (dropdown: Indoor/Outdoor, required).
  - **Watering Frequency** (number of days, required).
- Use **controlled components** to handle form inputs with React **state**.
- On form submission:
  - Send a POST request to the **JSON Server** to add the plant.
  - Redirect the user back to the **Plant List (Home Page)**.

3. Edit Plant Page

- Allow users to edit an existing plant:
  - Use **react-router-dom** to navigate to the **Edit Plant Page**.
  - Use **useParams** to retrieve the plant's ID from the URL.
  - Fetch the plant's details from the server and prefill the form with its current values.
  - Allow users to update the plant's name, type, and watering frequency.
  - Send a **PUT** request to the **JSON Server** to save the updated plant data.
  - Redirect the user back to the **Plant List** after updating the plant.

4. Search Plants

- Add a **search bar** to filter plants by their **name** dynamically.
- Use **state** to manage the search input. As the user types, the displayed list of plants should filter in real time to show only those matching the search query.

5. Global Filter (Indoor vs Outdoor Plants)

- Use **React Context** to create a global state for a filter that toggles between **Indoor** and **Outdoor** plants.
- Implement a toggle or dropdown menu on the Home Page to switch between these two categories.
- Display only the plants that match the selected category.

6. Styling

- Use **styled-components** for all styling.
- Design the Plant List page with visually appealing plant cards that include the name, type, and watering schedule.
- Add hover effects, responsive design, and simple animations to enhance the UI.
- Style the form for adding or editing plants to make it user-friendly and modern.

7. Navigation

- Use **react-router-dom** for routing between pages:
    - **Plant List (Home Page)**: Displays the list of plants.
    - **Add Plant Page**: Allows users to add a new plant.
    - **Edit Plant Page**: Allows users to edit an existing plant.
- Include navigation links or buttons to move between pages.

8. Data Storage

- Use **JSON Server** as a mock backend to store and manage plant data.
    - Create a db.json file with sample data for plants.

9. Deployment

- Deploy your **Gardening Planner App** to **Vercel**.

Deliverables

1. A **fully functional Gardening Planner App** hosted on **Vercel**.

Team 8:

Problem Statement: Build an Air Ticket Booking App

You have **3 hours** to build an **Air Ticket Booking Application** that allows users to search for flights, view available flights, and book tickets. This app will allow users to filter flights based on departure location, arrival location, date, and more.

Features & Requirements

1. Flight Search Page (Home Page)

- **Search Form**:
    - Create a search form that allows users to input:
        - **Departure Location** (dropdown or text input).
        - **Arrival Location** (dropdown or text input).
        - **Departure Date** (date picker).
    - **Search Button**: Once users fill in the form and click "Search," display available flights based on the entered details.
- **Flight Listings**:
    - Display a list of flights based on the search results (e.g., flight number, airline, departure time, price).
    - If no flights match, display the message: "No flights found" (conditional rendering).

2. Flight Booking Page

- **Flight Details**:
    - Show detailed information for the selected flight (e.g., flight number, airline, departure time, price).
- **Passenger Information Form**:
    - Allow the user to fill in their details:
        - **Full Name** (text input).
        - **Email** (email input).
        - **Phone Number** (text input).
    - **Submit Booking**:
        - Upon submitting, simulate the booking process (for now, just show a confirmation message: "Booking Successful").

3. Filter and Sort Options

- **Sort Flights**:
    - Allow users to sort available flights based on:
        - **Price** (ascending/descending).
        - **Departure Time** (earliest/latest).
- **Filter Flights**:
    - Allow users to filter by **airline**, **price range**, or **flight duration**.

4. Flight Listings

- **List Rendering**:

    - Render the list of available flights dynamically based on the search and filter criteria.
- **Flight Card**:
    - Each flight card should include details like:
        - **Flight Number**
        - **Airline Name**
        - **Departure Time**
        - **Price**
        - **Duration**

5. React Context for Global State

- **User Preferences**:
    - Use **React Context** to manage global state such as the user's preferred currency for flight prices or preferred sorting options.

6. Styling & Deployment

- **Styling**:
    - Use **styled-components** to style the search form, flight cards, and buttons.
    - Ensure a clean and responsive design for all screen sizes.
- **Deployment**:
    - Deploy the application to **Vercel** for easy online access.

Implementation Notes

- **Flight Data**:
    - Use **JSON Server** as a mock API to serve flight data (you can create an array of mock flight objects with properties like flight number, airline, departure date, price, etc.).
    - Use **useEffect** to simulate fetching data from the server.
    - Set up routes using **react-router-dom** to manage navigation between pages (Home, Booking Page).

Deliverables

1. A **fully functional Air Ticket Booking App** deployed to **Vercel**.
2. **Code clarity**: Ensure the code is clean, well-commented, and follows best practices.