1. \*\*Props\*\*:

- \*\*Objective\*\*: Create a `UserProfile` component that displays user details.

- \*\*Steps\*\*:

1. Define a `UserProfile` component that accepts `name` (string) and `age` (number) as props.

2. Display the user's name and age in a styled card format.

3. Use default messages "Name not provided" or "Age not provided" if `name` or `age` props are missing.

4. Ensure the component is reusable by accepting different `name` and `age` values.

2. \*\*State\*\*:

- \*\*Objective\*\*: Build a counter component with increment and decrement functionality.

- \*\*Steps\*\*:

1. Define a `Counter` component.

2. Use the `useState` hook to manage the `count` state, initialized to 0.

3. Create "Increment" and "Decrement" buttons.

4. Update the `count` state appropriately when the buttons are clicked.

5. Display the current count value in a styled manner.

3. \*\*Default Props\*\*:

- \*\*Objective\*\*: Implement a `Button` component with default styling.

- \*\*Steps\*\*:

1. Define a `Button` component that accepts `color` (string) and `size` (string) as props.

2. Set default props: `color` to 'blue' and `size` to 'medium'.

3. Style the button based on the `color` and `size` props.

4. Ensure the button renders correctly with both default and provided props.

4. \*\*React Components\*\*:

- \*\*Objective\*\*: Create a `TodoList` component that displays a list of todo items.

- \*\*Steps\*\*:

1. Define a `TodoList` component that accepts `todos` (array of objects with `id` and `text` properties) as props.

2. Map over the `todos` array and render each todo item in a list.

3. Style the list and each todo item for better presentation.

4. Ensure each todo item is uniquely identifiable using the `id` property.

5. \*\*Event Handling\*\*:

- \*\*Objective\*\*: Build a search form that alerts the input value on submission.

- \*\*Steps\*\*:

1. Define a `SearchForm` component.

2. Include an input field and a submit button.

3. Use the `useState` hook to manage the input value.

4. Add an `onSubmit` handler to the form that displays an alert with the input value.

5. Prevent the default form submission behavior using `event.preventDefault()`.

6. \*\*Conditional Rendering (&&)\*\*:

- \*\*Objective\*\*: Display a loading message based on a prop.

- \*\*Steps\*\*:

1. Define a `LoadingIndicator` component that accepts `loading` (boolean) as a prop.

2. Use the logical AND (&&) operator to conditionally render a "Loading..." message if `loading` is true.

3. Render the children elements if `loading` is false.

7. \*\*Conditional Rendering (||)\*\*:

- \*\*Objective\*\*: Show a message when a list is empty.

- \*\*Steps\*\*:

1. Define an `ItemList` component that accepts `items` (array) as a prop.

2. Use the logical OR (||) operator to display a "No items found" message if the `items` array is empty.

3. Render the list of items if the `items` array is not empty.

4. Style the list and message appropriately.

8. \*\*Conditional Rendering (Ternary Operator)\*\*:

- \*\*Objective\*\*: Display login/logout buttons based on authentication status.

- \*\*Steps\*\*:

1. Define an `AuthButton` component that accepts `isLoggedIn` (boolean) as a prop.

2. Use the ternary operator to conditionally render a "Login" button if `isLoggedIn` is false, or a "Logout" button if `isLoggedIn` is true.

3. Style the buttons for better user experience.

9. \*\*useRef Hook\*\*:

- \*\*Objective\*\*: Focus on an input field when the component mounts.

- \*\*Steps\*\*:

1. Define a `FocusInput` component.

2. Include an input field in the component.

3. Use the `useRef` hook to create a reference to the input field.

4. Use the `useEffect` hook to focus on the input field when the component mounts.

5. Ensure the input field is correctly focused by checking the browser behavior.

10. \*\*Higher Order Components (HOC)\*\*:

- \*\*Objective\*\*: Create a HOC that logs props to the console.

- \*\*Steps\*\*:

1. Define a `withLogger` HOC that wraps a component and logs its props to the console.

2. The HOC should accept a component as an argument and return a new component.

3. In the returned component, use `useEffect` to log the props to the console whenever they change.

4. Wrap a `SimpleComponent` with `withLogger` and verify the logging functionality by rendering `SimpleComponent` with different props.

11. \*\*Context API\*\*:

- \*\*Objective\*\*: Create a theme context and consume it in a component.

- \*\*Steps\*\*:

1. Define a `ThemeContext` with `light` and `dark` themes.

2. Create a `ThemeProvider` component that provides the `ThemeContext` value.

3. Define a `ThemedButton` component that consumes the `ThemeContext` and applies the appropriate styles based on the current theme.

4. Use the `ThemeProvider` to wrap a parent component and toggle between `light` and `dark` themes.

12. \*\*Lifecycle Methods in Function-Based Components\*\*:

- \*\*Objective\*\*: Fetch data when a component mounts.

- \*\*Steps\*\*:

1. Define a `FetchData` component.

2. Use the `useEffect` hook to simulate `componentDidMount`.

3. Inside `useEffect`, fetch data from an API when the component mounts.

4. Store the fetched data in a state variable and display it in a list.

5. Ensure proper error handling and loading state management.

13. \*\*useEffect Hook\*\*:

- \*\*Objective\*\*: Fetch and display user data from an API.

- \*\*Steps\*\*:

1. Define a `UserList` component.

2. Use the `useEffect` hook to fetch user data from an API when the component mounts.

3. Store the fetched data in a state variable.

4. Map over the fetched data and display it in a styled list.

5. Handle loading and error states appropriately.

14. \*\*React Routing\*\*:

- \*\*Objective\*\*: Set up basic routing for a multi-page application.

- \*\*Steps\*\*:

1. Install and configure `react-router-dom`.

2. Define `Home`, `About`, and `Contact` components with basic content.

3. Set up routing in the main `App` component to render these pages based on the URL.

4. Create a `Navbar` component with links to `Home`, `About`, and `Contact` pages.

5. Ensure that the appropriate page content is displayed when a link is clicked.

15. \*\*useReducer Hook\*\*:

- \*\*Objective\*\*: Implement a counter with `useReducer`.

- \*\*Steps\*\*:

1. Define a `CounterWithReducer` component.

2. Use `useReducer` to manage the counter state with an initial value of 0.

3. Define actions for "increment" and "decrement".

4. Create "Increment" and "Decrement" buttons that dispatch the respective actions.

5. Display the current count value.

16. \*\*useMemo\*\*:

- \*\*Objective\*\*: Optimize a component with an expensive calculation.

- \*\*Steps\*\*:

1. Define an `ExpensiveCalculation` component.

2. Include a computation function that takes significant time to execute.

3. Use `useMemo` to memoize the result of the computation, providing the computation function and its dependencies.

4. Display the result of the computation.

5. Include a way to trigger the computation, such as a button click.

17. \*\*useCallback\*\*:

- \*\*Objective\*\*: Memoize a callback function to prevent unnecessary re-renders.

- \*\*Steps\*\*:

1. Define a `ParentComponent` that renders a `ChildComponent`.

2. Pass a callback function from the parent to the child.

3. Use `useCallback` to memoize the callback function, providing the function and its dependencies.

4. Ensure the child component does not re-render unnecessarily by checking the console logs or using `React.memo`.

18. \*\*Redux Toolkit\*\*:

- \*\*Objective\*\*: Set up a Redux store and manage user authentication state.

- \*\*Steps\*\*:

1. Install and configure Redux Toolkit.

2. Create a slice for managing user authentication state with actions for login and logout.

3. Define initial state with properties like `isAuthenticated` and `user`.

4.

Implement a `LoginForm` component that dispatches the login action and updates the Redux state.

5. Use `useSelector` to access authentication state and conditionally render content based on the user's authentication status.

19. \*\*Props\*\*:

- \*\*Objective\*\*: Create a product card component that displays product details.

- \*\*Steps\*\*:

1. Define a `ProductCard` component that accepts `product` (object with `name`, `price`, and `description` properties) as a prop.

2. Display the product's name, price, and description in a styled card format.

3. Ensure the component is reusable with different product details.

20. \*\*State\*\*:

- \*\*Objective\*\*: Build a toggle switch component to manage on/off state.

- \*\*Steps\*\*:

1. Define a `ToggleSwitch` component.

2. Use the `useState` hook to manage the on/off state, initialized to false.

3. Create a button to toggle the state between "On" and "Off".

4. Display the current state ("On" or "Off") and style the button appropriately.

21. \*\*Default Props\*\*:

- \*\*Objective\*\*: Implement a notification component with default styling.

- \*\*Steps\*\*:

1. Define a `Notification` component that accepts `type` (string) and `message` (string) as props.

2. Set default props: `type` to 'info' and `message` to 'No new notifications'.

3. Style the notification based on the `type` prop, using different colors or icons for different types.

4. Ensure the notification renders correctly with both default and provided props.

22. \*\*React Components\*\*:

- \*\*Objective\*\*: Create a responsive navbar component with links.

- \*\*Steps\*\*:

1. Define a `Navbar` component.

2. Include links to different sections of a webpage, such as "Home", "About", "Services", and "Contact".

3. Style the navbar to be responsive, using CSS flexbox or grid.

4. Ensure the navbar looks good on different screen sizes and devices.

23. \*\*Event Handling\*\*:

- \*\*Objective\*\*: Build a dropdown menu with open/close functionality.

- \*\*Steps\*\*:

1. Define a `DropdownMenu` component.

2. Include a button to open and close the dropdown.

3. Use the `useState` hook to manage the dropdown's open/close state.

4. Display the dropdown menu items when the dropdown is open.

5. Ensure the dropdown menu is styled and accessible.

24. \*\*Conditional Rendering (&&)\*\*:

- \*\*Objective\*\*: Display an error message conditionally.

- \*\*Steps\*\*:

1. Define an `ErrorMessage` component that accepts `error` (boolean) as a prop.

2. Use the logical AND (&&) operator to conditionally render an error message if `error` is true.

3. Ensure the component renders nothing if `error` is false.

4. Style the error message for better visibility.

25. \*\*Conditional Rendering (||)\*\*:

- \*\*Objective\*\*: Display a placeholder text if the input value is empty.

- \*\*Steps\*\*:

1. Define a `PlaceholderInput` component that accepts `value` (string) as a prop.

2. Use the logical OR (||) operator to display a placeholder text if `value` is an empty string.

3. Display the input value if `value` is provided.

4. Ensure the input field is styled and behaves correctly.

26. \*\*Conditional Rendering (Ternary Operator)\*\*:

- \*\*Objective\*\*: Toggle between truncated and full text display.

- \*\*Steps\*\*:

1. Define a `TextToggle` component that accepts `text` (string) and `maxLength` (number) as props.

2. Display the text truncated to `maxLength` characters.

3. Include a "Read more" link if the text is truncated.

4. Display the full text and a "Read less" link when "Read more" is clicked.

5. Use the ternary operator to conditionally render the truncated or full text based on the state.

27. \*\*useRef Hook\*\*:

- \*\*Objective\*\*: Create an image carousel with navigation buttons.

- \*\*Steps\*\*:

1. Define an `ImageCarousel` component that accepts `images` (array of image URLs) as a prop.

2. Use `useRef` to keep track of the current slide.

3. Include next and previous buttons to navigate through the images.

4. Update the current slide using `useRef` when the buttons are clicked.

5. Display the current image and ensure the carousel is styled for a better user experience.

28. \*\*Higher Order Components (HOC)\*\*:

- \*\*Objective\*\*: Create a HOC that displays a loading spinner.

- \*\*Steps\*\*:

1. Define a `withLoadingSpinner` HOC that wraps a component and displays a loading spinner when a `loading` prop is true.

2. The HOC should accept a component as an argument and return a new component.

3. In the returned component, conditionally render the loading spinner based on the `loading` prop.

4. Wrap a `DataComponent` with `withLoadingSpinner` and verify the loading spinner functionality by toggling the `loading` prop.

29. \*\*Context API\*\*:

- \*\*Objective\*\*: Manage user authentication state with context.

- \*\*Steps\*\*:

1. Define an `AuthContext` with `isAuthenticated` and `login`/`logout` functions.

2. Create an `AuthProvider` component that provides the context value.

3. Define a `ProtectedRoute` component that checks the authentication state and conditionally renders its children or redirects to a login page.

4. Use the `AuthProvider` to wrap the main `App` component.

5. Implement a basic authentication flow to test the protected routes.

30. \*\*Lifecycle Methods in Function-Based Components\*\*:

- \*\*Objective\*\*: Simulate `shouldComponentUpdate` using `useEffect` and comparison logic.

- \*\*Steps\*\*:

1. Define a `Profile` component that accepts `user` (object) as a prop.

2. Use `useEffect` to perform a deep comparison of the `user` prop.

3. Only re-render the component if the `user` prop changes.

4. Display the user's profile information.

5. Ensure the component does not re-render unnecessarily by checking the console logs or using profiling tools.

31. \*\*useEffect Hook\*\*:

- \*\*Objective\*\*: Set up and clear an interval timer.

- \*\*Steps\*\*:

1. Define a `Clock` component.

2. Use the `useEffect` hook to set up an interval timer that updates every second.

3. Display the current time in a styled format.

4. Ensure the interval timer is cleared when the component unmounts to prevent memory leaks.

32. \*\*React Routing\*\*:

- \*\*Objective\*\*: Implement nested routing in a dashboard page.

- \*\*Steps\*\*:

1. Install and configure `react-router-dom`.

2. Define `Dashboard`, `Profile`, `Settings`, and `Statistics` components with basic content.

3. Set up routing in the main `App` component to render these pages based on the URL.

4. Create a navigation menu within the `Dashboard` page with links to `Profile`, `Settings`, and `Statistics`.

5. Ensure that the appropriate content is displayed when a link is clicked within the `Dashboard` page.

33. \*\*useReducer Hook\*\*:

- \*\*Objective\*\*: Manage a shopping cart state with `useReducer`.

- \*\*Steps\*\*:

1. Define a `ShoppingCart` component.

2. Use `useReducer` to manage the cart state with an initial value of an empty array.

3. Define actions for adding, removing, and clearing items in the cart.

4. Create functions to dispatch these actions and update the state.

5. Display the list of items in the cart and the total price.

6. Ensure the cart updates correctly when items are added, removed, or cleared.

34. \*\*useMemo\*\*:

- \*\*Objective\*\*: Memoize a filtered list based on a filter prop.

- \*\*Steps\*\*:

1. Define a `FilteredList` component that accepts `items` (array) and `filter` (string) as props.

2. Use `useMemo` to memoize the filtered list based on the `filter` prop.

3. Filter the `items` array based on the `filter` string and store the result in a memoized value.

4. Display the filtered list in a styled manner.

5. Ensure the filtering logic is optimized and only recalculates when the `filter` prop changes.

35. \*\*useCallback\*\*:

- \*\*Objective\*\*: Prevent unnecessary re-renders of a child component by memoizing a callback function.

- \*\*Steps\*\*:

1. Define a `ParentComponent` that renders a `ChildComponent`.

2. Pass a callback function from the parent to the child.

3. Use `useCallback` to memoize the callback function, providing the function and its dependencies.

4. In the child component, perform an action such as logging a message or updating the state when the callback is called.

5. Ensure the child component does not re-render unnecessarily by checking the console logs or using `React.memo`.

36. \*\*Redux Toolkit\*\*:

- \*\*Objective\*\*: Set up a Redux store and manage a todo list.

- \*\*Steps\*\*:

1. Install and configure Redux Toolkit.

2. Create a slice for managing a todo list with actions for adding, removing, and toggling todo items.

3. Define initial state with an array of todos.

4. Implement a `TodoList` component that connects to the Redux store and displays the todo list.

5. Create functions to dispatch actions for adding, removing, and toggling todos.

6. Ensure the todo list updates correctly when actions are dispatched.

37. \*\*Props\*\*:

- \*\*Objective\*\*: Create a `WeatherCard` component that displays weather details.

- \*\*Steps\*\*:

1. Define a `WeatherCard` component that accepts `weather` (object with `temperature`, `condition`, and `location` properties) as a prop.

2. Display the weather details in a styled card format, including temperature, condition, and location.

3. Ensure the component is reusable with different weather details.

38. \*\*State\*\*:

- \*\*Objective\*\*: Build a modal component with open/close functionality.

- \*\*Steps\*\*:

1. Define a `Modal` component.

2. Use the `useState` hook to manage the modal's open/close state with a boolean variable, initialized to false.

3. Include a button to open and close the modal.

4. Display the modal content when it is open, and ensure it is hidden when closed.

5. Style the modal and its content for a better user experience.

39. \*\*Default Props\*\*:

- \*\*Objective\*\*: Implement a `Badge` component with default color and text.

- \*\*Steps\*\*:

1. Define a `Badge` component that accepts `color` (string) and `text` (string) as props.

2. Set default props: `color` to 'grey' and `text` to 'New'.

3. Style the badge based on the `color` prop and display the `text` prop inside the badge.

4. Ensure the badge renders correctly with both default and provided props.

40. \*\*React Components\*\*:

- \*\*Objective\*\*: Create a responsive gallery component to display images.

- \*\*Steps\*\*:

1. Define a `Gallery` component that accepts `images` (array of image URLs) as a prop.

2. Map over the `images` array and render each image in a responsive grid layout.

3. Style the gallery to be responsive, using CSS flexbox or grid.

4. Ensure the gallery looks good on different screen sizes and devices.

41. \*\*Event Handling\*\*:

- \*\*Objective\*\*: Build a dropdown menu with open/close functionality.

- \*\*Steps\*\*:

1. Define a `DropdownMenu` component.

2. Include a button to open and close the dropdown.

3. Use the `useState` hook to manage the dropdown's open/close state.

4. Display the dropdown menu items when the dropdown is open.

5. Ensure the dropdown menu is styled and accessible.

42. \*\*Conditional Rendering (&&)\*\*:

- \*\*Objective\*\*: Display an error message conditionally.

- \*\*Steps\*\*:

1. Define an `ErrorMessage` component that accepts `error` (boolean) as a prop.

2. Use the logical AND (&&) operator to conditionally render an error message if `error` is true.

3. Ensure the component renders nothing if `error` is false.

4. Style the error message for better visibility.

43. \*\*Conditional Rendering (||)\*\*:

- \*\*Objective\*\*: Display a placeholder text if the input value is empty.

- \*\*Steps\*\*:

1. Define a `PlaceholderInput` component that accepts `value` (string) as a prop.

2. Use the logical OR (||) operator to display a placeholder text if `value` is an empty string.

3. Display the input value if `value` is provided.

4. Ensure the input field is styled and behaves correctly.

44. \*\*Conditional Rendering (Ternary Operator)\*\*:

- \*\*Objective\*\*: Toggle between truncated and full text display.

- \*\*Steps\*\*:

1. Define a `TextToggle` component that accepts `text` (string) and `maxLength` (number) as props.

2. Display the text truncated to `maxLength` characters.

3. Include a "Read more" link if the text is truncated.

4. Display the full text and a "Read less" link when "Read more" is clicked.

5. Use the ternary operator to conditionally render the truncated or full text based on the state.

45. \*\*useRef Hook\*\*:

- \*\*Objective\*\*: Create an image carousel with navigation buttons.

- \*\*Steps\*\*:

1. Define an `ImageCarousel` component that accepts `images` (array of image URLs) as a prop.

2. Use `useRef` to keep track of the current slide.

3. Include next and previous buttons to navigate through the images.

4. Update the current slide using `useRef` when the buttons are clicked.

5. Display the current image and ensure the carousel is styled for a better user experience.

46. \*\*Higher Order Components (HOC)\*\*:

- \*\*Objective\*\*: Create a HOC that displays a loading spinner.

- \*\*Steps\*\*:

1. Define a `withLoadingSpinner` HOC that wraps a component and displays a loading spinner when a `loading` prop is true.

2. The HOC should accept a component as an argument and return a new component.

3. In the returned component, conditionally render the loading spinner based on the `loading` prop.

4. Wrap a `DataComponent` with `withLoadingSpinner` and verify the loading spinner functionality by toggling the `loading` prop.

47. \*\*Context API\*\*:

- \*\*Objective\*\*: Manage user authentication state with context.

- \*\*Steps\*\*:

1. Define an `AuthContext` with `isAuthenticated` and `login`/`logout` functions.

2. Create an `AuthProvider` component that provides the context value.

3. Define a `ProtectedRoute` component that checks the authentication state and conditionally renders its children or redirects to a login page.

4. Use the `AuthProvider` to wrap the main `App` component.

5. Implement a basic authentication flow to test the protected routes.

48. \*\*Lifecycle Methods in Function-Based Components\*\*:

- \*\*Objective\*\*: Simulate `shouldComponentUpdate` using `useEffect` and comparison logic.

- \*\*Steps\*\*:

1. Define a `Profile` component that accepts `user` (object) as a prop.

2. Use `useEffect` to perform a deep comparison of the `user` prop.

3. Only re-render the component if the `user` prop changes.

4. Display the user's profile information.

5. Ensure the component does not re-render unnecessarily by checking the console logs or using profiling tools.

49. \*\*useEffect Hook\*\*:

- \*\*Objective\*\*: Set up and clear an interval timer.

- \*\*Steps\*\*:

1. Define a `Clock` component.

2. Use the `useEffect` hook to set up an interval timer that updates every second.

3. Display the current time in a styled format.

4. Ensure the interval timer is cleared when the component unmounts to prevent memory leaks.

50. \*\*React Routing\*\*:

- \*\*Objective\*\*: Implement protected routes using React Router.

- \*\*Steps\*\*:

1. Install and configure `react-router-dom`.

2. Define `Login`, `Dashboard`, and `Profile` components with basic content.

3. Set up routing in the main `App` component to render these pages based on the URL.

4. Create a `PrivateRoute` component that checks if the user is authenticated and renders the `Dashboard` or `Profile` component if authenticated, or redirects to the `Login` page if not.

5. Implement a basic authentication flow to test the protected routes.