Hooks:

React Hooks are a new feature of React.js that makes it possible to use state and other React features without writing a class. Hooks allow us to "hook" into React features such as state and lifecycle methods.

Hooks allow function components to have access to state and other React features. Because of this, class components are generally no longer needed. Hooks were introduced in React 16.8 as a way to write reusable and stateful logic without using class components.

1.Use State:

The useState hook in React is used to **add state functionality to functional components**. It allows you to declare and manage state variables within your component.

Example: Form Input handling, toggle components(On/Off), Drop downs,

```
import React, { useState } from 'react';

function Counter() {
// Declare a state variable called 'count' and initialize it to 0
  const [count, setCount] = useState(0);

  const increment = () => {
    // Update the 'count' state by increasing it by 1
    setCount(count + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default Counter;
```

2. UseEffect:

In React, the useEffect hook is a built-in function that allows you to perform side effects in functional components. Side effects refer to actions, such as **interacting with the browser (DOM), making HTTP requests, setting up timers, Data Fetching updates and clean ups.**

```
import React, { useState, useEffect } from "react";

const Counter = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    // This runs both on mount and on every update (componentDidMount + componentDidUpdate)
    const timer = setTimeout(() => {
      setCount((prevCount) => prevCount + 1);
    }, 1000);

    // This cleanup function runs on unmount (componentWillUnmount) and before every re-render
    return () => {
      clearTimeout(timer);
    };
  }, [count]); // Dependency array ensures the effect runs on mount and whenever 'count' changes

  return (
    <div>
      <p>Count: {count}</p>
    </div>
  );
};

export default Counter;
```

Dependency Array:

Here's how the dependency array works:

1) **Empty Dependency Array ([]):**

`useEffect(() => { // Effect code }, []);`

Behavior: The effect runs once after the initial render.

Use Case: This is useful when you want the effect to mimic componentDidMount, running only after the component mounts.

2) **Non-Empty Dependency Array:**

`useEffect(() => { // Effect code }, [dependency1, dependency2]);`

Behavior: The effect runs after the initial render and whenever any value in the dependency array changes between renders.

Use Case: This is useful when you want the effect to run based on changes in specific variables or state values.

3) <u>No Dependency Array:</u>

useEffect(() => { // Effect code });

Behavior: The effect runs after every render.

Use Case: Be cautious when using this approach, as it can lead to the effect running frequently, potentially causing performance issues. It's often better to specify dependencies to control when the effect runs.

Achieve Life cycle methods using useEffect:

componentDidMount Equivalent

- To replicate `componentDidMount`, you can use `useEffect` with an empty dependency array (`[]`). This ensures that the effect runs only once after the initial render, simulating the behavior of `componentDidMount`.

componentDidUpdate Equivalent

- To replicate `componentDidUpdate`, you can use `useEffect` with a dependency array containing the variables that you want to monitor for changes. The effect will then run every time any of these dependencies change.

componentWillUnmount Equivalent

- To replicate `componentWillUnmount`, you can return a cleanup function from `useEffect`. This cleanup function will run when the component is about to be unmounted.

Key Use Cases for useEffect:

1. **Fetching Data:**
    - Use useEffect to fetch data from an API or perform asynchronous operations when the component mounts or when certain dependencies change.

```
useEffect(() => {
  const fetchData = async () => {
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    setData(data);
  };
  fetchData();
}, []); // Empty dependency array means it runs once after initial
render
```

2. **Subscriptions:**

  - `useEffect` can manage subscriptions to external data sources (like WebSockets) or event listeners that need to be cleaned up when the component unmounts.

```
useEffect(() => {
  const subscription = externalAPI.subscribe(handleData);
  return () => {
    subscription.unsubscribe();
  };
}, [handleData]); // Dependency array ensures subscription is re-
established when handleData changes
```

3. UseContext:

The **useContext** hook in React is used to consume values from a context created by the **createContext** API. It allows you to pass data through the component tree without the need to pass props down manually at every level. It is especially useful for sharing data that is required by many components at different levels of the component tree.

It allows you to access and use context values within functional components without the need for prop drilling.

Real time scenarios are **Theme switching, User authentication, language and localization, Global state management.**

Example 1: Theme switching

```
import React, { createContext, useContext } from 'react';

// Create a context
const ThemeContext = createContext();

// A component that provides the context value
function App() {
  const theme = 'dark';

  return (
    <ThemeContext.Provider value={theme}>
      <ChildComponent />
    </ThemeContext.Provider>
  );
}

// A child component that consumes the context value
function ChildComponent() {
  const theme = useContext(ThemeContext);
  return <div>Current theme: {theme}</div>;
}

export default App;
```

Example 2: Language / Localization

```
import React, { createContext, useContext } from 'react';

// Create a Context
const LangContext = createContext('en');

function App() {
  return (
    <LangContext.Provider value='en'>
      <ChildComponent />
    </LangContext.Provider>
  );
}

function ChildComponent() {
  const lang = useContext(LangContext);

  return (
    <div>
      {lang === 'en' && <h1>Hello!</h1>}
      {lang === 'ta' && <h1>Vanakkam!</h1>}
    </div>
  );
}

export default App;
```

Example 3: User Authentication

```
import React, { createContext, useContext, useState } from "react";

// Create the AuthContext
const AuthContext = createContext();

// AuthProvider component
```

```
function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = () => {
    setUser({ username: "Obed" }); // Updated to correct object syntax
  };

  const logout = () => {
    setUser(null);
  };

  return (
    <AuthContext.Provider value={{ user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

// Custom hook to use the AuthContext
const useAuth = () => {
  return useContext(AuthContext);
};

// ChildComponent to consume the context
function ChildComponent() {
  const { user, login, logout } = useContext(AuthContext);
  // const { user, login, logout } = useAuth();

  return (
    <div>
      {user ? (
        <div>
          <p>Welcome, {user.username}</p>
          <button onClick={logout}>Log Out</button>
        </div>
      ) : (
        <button onClick={login}>Log In</button>
      )}
    </div>
  );
}
// Usage in App component
function App() {
  return (
    <AuthProvider>
      <ChildComponent />
    </AuthProvider>
  );
}

export default App;
```

4. useRef:

It is used to **create a mutable reference** that persists across renders of a functional component. It is used to access / store references to DOM elements, manage previous values of a state, or to store any value that needs to be retained between renders without triggering a re-render.

(i) Accessing/Referencing and Modifying DOM Elements:

```
import React, { useRef, useEffect } from 'react';

const App = () => {
  const inputRef = useRef(null);
  useEffect(() => {
    // Accessing the DOM element
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} />;
};
export default App;
```

(ii) Keeping Values across Renders without Triggering Re-renders:

```
import React, { useRef } from "react";

function Counter() {
  const counterRef = useRef(0);

  const increment = () => {
    counterRef.current += 1;
    console.log(counterRef.current);
  };

  return (
    <div>
      <p> Counter: {counterRef.current} </p>
      <button onClick={increment}> Increment </button>
    </div>
  );
}

export default Counter;
```

(iii) Storing Previous Values:

```
import React, { useRef, useEffect } from 'react';

function PreviousValueDisplay() {
  const previousValueRef = useRef(null); // Create a ref to store the previous value
  const currentValue = 13; // Define the current value

  useEffect(() => {
    previousValueRef.current = currentValue; // Update the ref with the current value
  }, [currentValue]); // Effect depends on currentValue

  return (
    <div>
      <p>Current Value: {currentValue}</p>
      <p>Previous Value: {previousValueRef.current}</p>
    </div>
  );
}

export default PreviousValueDisplay;
```

5. useReducer:

useReducer is a powerful hook in React used for managing state in functional components. It is an **alternative to useState** and provides a way to **handle more complex state logics** in the components. It is often used when the state transitions are more involved and need to be managed in a centralized manner.

The useReducer hook takes in a reducer function and an initial state, and returns the current state and a dispatch function to trigger state updates. The reducer function is responsible for specifying how the state should change in response to different actions.

Real time scenarios: (i) User authentication with login, logout, (ii) Manage Form states where changes in one field may affect others. (iii)Managing items in shopping carts where add, remove, update operations.

Example 1: Counter App with Increment & Decrement

```
import React, { useReducer } from "react";

const initialState = { count: 0 };

const reducer = (state, action) => {
  switch (action.type) {
    case "INCREMENT":
      return { count: state.count + 1 };
    case "DECREMENT":
      return { count: state.count - 1 };
    default:
      return state;
  }
};

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  const increment = () => {
    dispatch({ type: "INCREMENT" });
  };

  const decrement = () => {
    dispatch({ type: "DECREMENT" });
  };
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default Counter;
```

Example 2: User Authentication

```
import React, { createContext, useContext, useReducer } from "react";

// Define action types
const LOGIN = "LOGIN";
const LOGOUT = "LOGOUT";

// Define the reducer function
const authReducer = (state, action) => {
  switch (action.type) {
    case LOGIN:
      return { user: { username: "Obed" } };
    case LOGOUT:
      return { user: null };
    default:
      return state;
  }
};

// Create the AuthContext
const AuthContext = createContext();

// AuthProvider component
function AuthProvider({ children }) {
  const [state, dispatch] = useReducer(authReducer, { user: null });

  const login = () => dispatch({ type: LOGIN });
  const logout = () => dispatch({ type: LOGOUT });

  return (
    <AuthContext.Provider value={{ user: state.user, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}

// Custom hook to use the AuthContext
const useAuth = () => useContext(AuthContext);

// ChildComponent to consume the context
function ChildComponent() {
  const { user, login, logout } = useAuth();

  return (
    <div>
      {user ? (
        <div>
          <p>Welcome, {user.username}</p>
          <button onClick={logout}>Log Out</button>
        </div>
      ) : (
        <button onClick={login}>Log In</button>
      )}
    </div>
  );
}

// App component
function App() {
  return (
    <AuthProvider>
      <ChildComponent />
    </AuthProvider>
  );
}

export default App;
```

6 & 7: UseCallback and useMemo Hooks:

The **useCallback** hook is used to memoize functions in React. When a function is wrapped with **useCallback**, it will only be recreated if one of its dependency's changes. This is particularly useful when passing functions down to child components to prevent unnecessary re-renders.

The **useMemo** hook is used to memoize the results of a computation. It's particularly useful when you have a costly calculation that you want to avoid re-running on every render.

Instead, you can use **useMemo** to store the result of the computation and only recompute it if the dependencies have changed.

```
import React, { useState, useCallback, useMemo } from 'react';

function App() {
  const [count, setCount] = useState(0);

  // Increment function wrapped with useCallback
  const increment = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  // Calculate square of 'count' using useMemo
  const square = useMemo(() => {
    console.log('Calculating square...');
    return count * count;
  }, [count]); //(square) should only be recalculated when count changes

  return (
    <div>
      <p>Count: {count}</p>
      <p>Square: {square}</p>
      <button onClick={increment}>Increment Count</button>
    </div>
  );
}

export default App;
```

- `useCallback` to memoize functions to prevent unnecessary re-renders.
- `useMemo` is used to calculate the square of `count` and memoize its result in the `square` variable. The dependency array `[count]` specifies that the memoized value (square) should only be recalculated when count changes.

Example 2:

```
import React, { useState, useCallback, useMemo } from 'react';

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  // useCallback to memoize increment function
  const increment = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  // useCallback to memoize addTodo function
  const addTodo = useCallback(() => {
    setTodos((prevTodos) => [...prevTodos, 'New Todo']);
  }, []);

  // useMemo to memoize an expensive calculation
  const square = useMemo(() => {
    console.log('Calculating...');
    return count * count; // Simple example of an expensive calculation
  }, [count]);

  return (
    <div>
      <div>
        <h2>My Todos</h2>
        {todos.map((todo, index) => (
          <p key={index}>{todo}</p>
        ))}
        <button onClick={addTodo}>Add Todo</button>
      </div>
      <hr />
      <div>
        <h2>Counter</h2>
        <p>Count: {count}</p>
        <p>Square: {square}</p>
        <button onClick={increment}>Increment</button>
      </div>
    </div>
  );
};

export default App;
```

useMemo vs **React.memo:**

useMemo and **React.memo** are both tools in React that help optimize performance by memoizing values or preventing unnecessary renders. However, they serve different purposes and are used in different contexts.

useMemo:

Hook Purpose:

useMemo is a hook in React used to memoize the result of a computation. It's primarily used to memoize values and prevent the recalculation of expensive computations on every render.

Usage:

It takes two arguments: a function that computes a value, and an array of dependencies. The value is only recomputed if one of the dependencies has changed.

Example:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

React.memo:

Higher-Order Component:

React.memo is a higher-order component (HOC) used to memoize functional components. It prevents a functional component from re-rendering if its props haven't changed.

Usage:

It's used by wrapping a functional component. The wrapped component will be re-rendered only if its props change.

Example:

```
const MyComponent = React.memo((props) => {
  // Component logic
});
```

Key Differences:

Purpose:

useMemo is used to memoize values within a component function.

React.memo is used to memoize the entire functional component.

Usage Context:

Use useMemo when you want to memoize values or avoid unnecessary computations within the body of a functional component.

Use React.memo when you want to prevent a functional component from re-rendering if its props have not changed.

Dependencies:

useMemo takes an array of dependencies to determine when to recalculate the memoized value.

React.memo automatically checks all props for changes.

Example 1:

```
import React, { useState, useCallback, useMemo } from 'react';

const Display = React.memo(({ count, square }) => {
  return (
    <div>
      <p>Count: {count}</p>
      <p>Square: {square}</p>
    </div>
  );
});

function App() {
  const [count, setCount] = useState(0);

  // Increment function wrapped with useCallback
  const increment = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  // Calculate square of 'count' using useMemo
  const square = useMemo(() => {
    return count * count;
  }, [count]); //(square) should only be recalculated when count changes

  return (
    <div>
      <Display count={count} square={square} />
      <button onClick={increment}>Increment Count</button>
    </div>
  );
}
```

```
export default App;
```

Example 2:

```
import React, { useState, useCallback, useMemo } from 'react';

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  // useCallback to memoize increment function
  const increment = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  // useCallback to memoize addTodo function
  const addTodo = useCallback(() => {
    setTodos((prevTodos) => [...prevTodos, 'New Todo']);
  }, []);

  // useMemo to memoize an expensive calculation
  const square = useMemo(() => {
    return count * count; // Simple example of an expensive calculation
  }, [count]);

  return (
    <div>
      <Todos todos={todos} addTodo={addTodo} />
      <h2>Counter</h2>
      <button onClick={increment}>Increment</button>
      <p>Count: {count}</p>
      <p>Square: {square}</p>
    </div>
  );
};

// Memoized Todos component using React.memo
const Todos = React.memo(({ todos, addTodo }) => {
  return (
    <div>
```

```
      <h2>My Todos</h2>
      <button onClick={addTodo}>Add Todo</button>
      {todos.map((todo, index) => (
        <p key={index}>{todo}</p>
      ))}
    </div>
  );
});

export default App;
```

Higher Order Component:

A Higher-Order Component (HOC) is a pattern in React used to reuse component logic. An HOC is a function that takes a component and returns a new component with additional properties or behaviour. It is a form of component composition in React that allows you to share functionality between components without duplicating code.

Key Characteristics:

- **Function**: An HOC is a function that takes a component as an argument and returns a new component.
- **Reusability**: It enables the reuse of component logic across multiple components.
- **Pure Function**: HOCs are pure functions—they do not modify the original component but return a new component with enhanced features.

```
import React from 'react';

// Higher-Order Component that provides theme props to the wrapped component
const WithTheme = (WrappedComponent) => {
  return (props) => {
    const theme = { color: 'blue', background: 'lightgray' };

    return <WrappedComponent theme={theme} {...props} />;
  };
};

// Simple component that uses the theme
const ThemedComponent = ({ theme, text }) => {
  return (
    <div style={{ color: theme.color, background: theme.background,
padding: '10px' }}>
      <p>{text}</p>
    </div>
  );
};

// Enhance ThemedComponent with withTheme HOC
const EnhancedComponent = WithTheme(ThemedComponent);

// App component that uses the enhanced component
const App = () => {
  return (
    <div>
      <EnhancedComponent text="This is a themed component!" />
    </div>
  );
};

export default App;
```

Pure Functions & Pure Functional Component:

Pure function is a function that always return the same result given the same inputs and have no side effects. They contribute to making React components more predictable, maintainable, and performant.

Ex:

```
function add(a,b) {
  return a+b;
}
```

Pure Component is a component that only depends on its props and state and does not have side effects. It renders the same output given the same props and state.

For functional components, you use React.memo to create a "pure" functional component. React.memo is a higher-order component that memoizes the result of a functional component. It prevents unnecessary re-renders by performing a shallow comparison of the component's props.

```
import React, { useState } from 'react';

// Functional component wrapped with React.memo
const PureFunctionalComponent = React.memo(({ text }) => {
  return <p>{text}</p>;
});
```

```
const App = () => {
  const [text, setText] = useState('Hello');

  return (
    <div>
      <PureFunctionalComponent text={text} />
      <button onClick={() => setText(text === 'Hello' ? 'Hello World' :
'Hello')}>
        Toggle Text
      </button>
    </div>
  );
};

export default App;
```

Custom hook:

Custom hooks are a powerful feature in React that help in organizing, reusing, and managing logic across your components. They provide a way to share stateful logic without changing the component hierarchy, keeping your components clean and focused on rendering.

```
// useCounter.js
import { useState } from 'react';

const useCounter = () => {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
  };

  return { count, increment };
};

export default useCounter;
```

```
import React from 'react';
import useCounter from './useCounter';

const CounterComponent = () => {
  const { count, increment, decrement } = useCounter(0, 2);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default CounterComponent;
```

{ ...state, count: state.count + 1 } Vs { count: state.count + 1 }

1. return { ...state, count: state.count + 1 };

Usage:

- This approach is used when you want to update the count property of the state object (state) while keeping all other properties unchanged.

- It spreads the current state (...state) into a new object and then overrides the count property with the updated value `(state.count + 1)`.

2. return { count: state.count + 1 };

Usage:

- This approach is used when you only need to return the updated property (count in this case) without preserving any other properties from the current state.

- It directly creates a new object with only the count property updated to state.count + 1.

When to Use Each:

- *First Approach*: Use when you want to update a specific property (count) while retaining other properties (...state). It ensures that you maintain the integrity of the entire state object.

- *Second Approach*: Use when you only need to update a specific property (count) and do not need to consider or retain any other properties from the current state.

Life Cycle Methods of Class Component:

Mounting Phases:

1. *constructor(props):*

   - The constructor method is called before a component is mounted.

   - It is used for initializing state and binding event handlers.

   - Note: Avoid using setState in the constructor; instead, initialize state directly.

2. *componentWillMount (deprecated):*

   - This method is deprecated and should not be used. It was called just before the component is rendered for the first time.

3. *render:*

   - The render method is required and returns the JSX (or null/undefined) that represents the component's UI.

   - It should be a pure function, meaning it should not alter component state or interact with the browser.

   - This method should be kept free of side effects.

4. *componentDidMount:*

   - componentDidMount is called immediately after the component is mounted (inserted into the DOM).

   - It is commonly used for initialization that requires DOM nodes or data fetching (e.g., AJAX requests).

Updating Phase:

5. *componentWillReceiveProps(nextProps) (deprecated):*

   - Deprecated and replaced with static getDerivedStateFromProps.

6. *shouldComponentUpdate(nextProps, nextState):*

- This method allows you to control if the component should re-render on prop or state change.

- It returns a boolean value (true for re-render, false to skip re-render).

7. *componentWillUpdate(nextProps, nextState) (deprecated):*

- Deprecated and replaced with getSnapshotBeforeUpdate.

8. *getSnapshotBeforeUpdate(prevProps, prevState):*

- getSnapshotBeforeUpdate is called right before the changes from the virtual DOM are to be reflected in the real DOM.

- It is often used to capture information (e.g., scroll position) before an update and return it for the componentDidUpdate method.

9. *componentDidUpdate(prevProps, prevState, snapshot):*

- componentDidUpdate is called immediately after updating occurs. This method is not called for the initial render.

Unmounting Phase:

10. *componentWillUnmount():*

- componentWillUnmount is called immediately before the component is unmounted (removed from the DOM).

- It is used for cleanup tasks such as cancelling network requests, timers, or removing event listeners.

Error Handling Phase:

11. *static getDerivedStateFromError(error):*

- This lifecycle method is called when an error is thrown in a child component.

- It is called during the "render" phase, so side-effects are not permitted.

12. *componentDidCatch(error, info):*

- This lifecycle method is called after an error has been thrown by a descendant component. It is used to log error information and display a fallback UI.

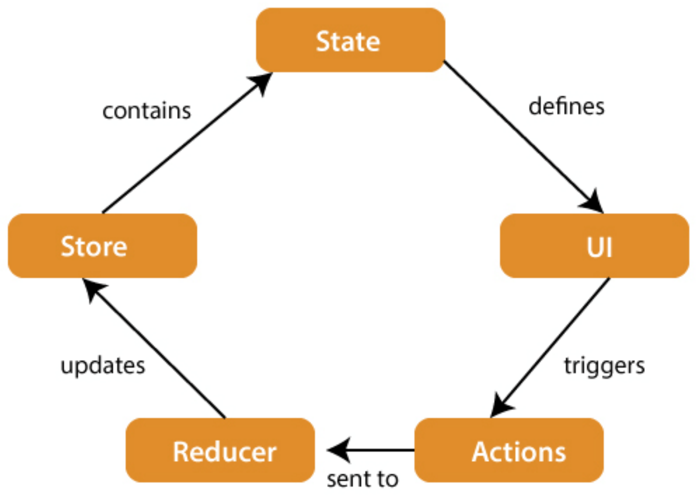
Redux:

Redux is a state management library commonly used with React applications. It helps manage the state of your application in a predictable and centralized way.

Redux is a predictable state container for JavaScript applications, commonly used with libraries like React or Angular. It provides a centralized place to manage the state of your application, making state management more predictable and easier to debug.

Concepts in Redux:

1. **State**: In Redux, the entire application state is stored in a single plain JavaScript object called the **store**. This makes it easy to access and manage the state of your application from a centralized location.
2. **Actions**: Actions are payloads of information that send data from your application to your Redux store. They are plain JavaScript objects that must have a `type` property indicating the type of action being performed. Actions are typically created using action creators, which are functions that return action objects.
3. **Reducers**: Reducers specify how the application's state changes in response to actions sent to the store. A reducer is a pure function that takes the previous state and an action, and returns the next state of the application. It determines what changes to make to the state based on the action type.
4. **Store**: The store is the object that brings actions, reducers, and state together. It holds the application state and allows access to state via `getState`(), allows state to be updated via `dispatch(action)`, and registers listeners via `subscribe(listener)`.
5. **Middleware**: Middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer. It is commonly used for logging actions, performing asynchronous tasks like fetching data, or routing actions to different parts of the application.



Workflows in Redux:

1. Defining State:

- **State Shape**: Design the structure of your application's state as a plain JavaScript object. This is often called the initial state and is defined in the reducers.

2. Creating Actions:

- **Action Types**: Define constants for different action types (`INCREMENT`, `DECREMENT`, etc.).
- **Action Creators**: Functions that create and return action objects. These functions encapsulate the logic of creating actions with predefined types and payload data.

3. Writing Reducers:

- **Reducer Function**: Write a pure function (the reducer) that takes the current state and an action, and returns a new state based on the action type. It should never mutate the state directly, but return a new object representing the next state.

4. Setting Up the Store:

- **Create Store**: Use `createStore()` from Redux to create the store, passing in the root reducer. This initializes the application state based on the reducers provided.

5. Connecting Components:

- **Provider**: Wrap the root component of your application with `<Provider>` from `react-redux`. This provides the Redux store to all components in the application.
- **useSelector**: Hook provided by `react-redux` to extract data from the Redux store state in functional components.

- **useDispatch**: Hook provided by `react-redux` to dispatch actions from functional components.

6. Dispatching Actions:

- **Dispatch Actions**: Components dispatch actions using `dispatch()` function obtained from `useDispatch()`. Actions are dispatched to the store, which triggers the appropriate reducers to update the state.

7. Updating UI:

- **State Updates**: When reducers update the state, the components that are connected to the relevant parts of the state via `useSelector()` automatically re-render with the updated data.
- **Immutable Updates**: Redux encourages immutability. Reducers should always return new objects (or arrays) rather than mutating the existing state.

8. Handling Asynchronous Actions:

- **Middleware**: For handling asynchronous actions like fetching data from an API, Redux middleware like `redux-thunk` or `redux-saga` can be used. Middleware intercepts dispatched actions before they reach the reducers, allowing for async logic.

Benefits of Using Redux:

- **Centralized State**: All application state is kept in a single store, making it easier to manage and debug.
- **Predictable State Changes**: State mutations are centralized and happen in a predictable manner via reducers, ensuring a consistent application state.
- **Debugging**: Redux's developer tools provide powerful debugging capabilities, such as time-travel debugging and state snapshotting.
- **Ecosystem**: Redux has a large ecosystem of tools and extensions (like middleware) that can extend its capabilities for various use cases.

Example of Redux:

First, you'll need to install Redux and its related packages. In your project directory, run the following command:

npm install redux react-redux

```
import React from "react";
import { createStore } from "redux";
import { Provider, useDispatch, useSelector } from "react-redux";
```

```
// Redux actions
const INCREMENT = "INCREMENT";
const DECREMENT = "DECREMENT";

// Redux action creators
const increment = () => ({ type: INCREMENT });
const decrement = () => ({ type: DECREMENT });

// Redux reducer
const counterReducer = (state = { count: 0 }, action) => {
  switch (action.type) {
    case INCREMENT:
      return { count: state.count + 1 };
    case DECREMENT:
      return { count: state.count - 1 };
    default:
      return state;
  }
};

// Create Redux store
const store = createStore(counterReducer);

const CounterApp = () => {
  const count = useSelector((state) => state.count); // Get the current
count from the store
  const dispatch = useDispatch(); // Get the dispatch function

  const handleIncrement = () => {
    dispatch(increment());
  };

  const handleDecrement = () => {
    dispatch(decrement());
  };

  return (
    <div style={{ textAlign: "center", marginTop: "50px" }}>
      <h1>Counter Application with Redux</h1>
      <h2>Count: {count}</h2>
      <button onClick={handleIncrement}>Increment</button>
      <button onClick={handleDecrement}>Decrement</button>
    </div>
  );
};

const App = () => (
  <Provider store={store}>
    <CounterApp />
  </Provider>
);

export default App;
```

3 principles of Redux:

Single Source of Truth:

- All state in one store.
- Easier to debug and manage.

State is Read-Only:

- Change state only by dispatching actions.
- Prevents accidental changes and helps track why changes happen.

Changes are Made with Pure Reducer Functions:

- Reducers decide how state changes based on actions.
- Pure functions make state changes predictable and easier to test.

Prop Drilling:

Prop drilling refers to the process of passing props (properties) from a parent component through one or more intermediate components down to a child component that needs access to the prop. It can occur when components need to share data or functionality that is not directly related to the parent-child relationship.

```
import React from 'react';

// Parent component
function ParentComponent() {
  const data = "Some data";
  return (
    <div>
      <IntermediateComponent data={data} />
    </div>
  );
}
```

```
// Intermediate component
function IntermediateComponent({ data }) {
  return (
    <div>
      <ChildComponent data={data} />
    </div>
  );
}

// Child component
function ChildComponent({ data }) {
  return <div>{data}</div>;
}

export default ParentComponent;
```

In the example above, the ParentComponent has some data that needs to be accessed by the ChildComponent. However, since IntermediateComponent is in between, the data prop needs to be passed through it to reach ChildComponent. This is known as prop drilling.

Prop drilling can lead to several issues:

- This reduces component reusability and increases dependencies.
- When props change at the top level, all intermediate components in the prop drilling chain receive the updated props, potentially causing unnecessary re-renders and impacting performance.
- To mitigate the issues associated with prop drilling, various state management solutions like Redux, React Context, or custom hooks can be used. These solutions allow for a more centralized and accessible state without the need to pass props through every intermediate component. They provide a way to share data or functionality across components efficiently and decouple components from the specific prop requirements of their parents.

State Lifting:

State lifting in React refers to the practice of moving the state of a component higher up the component tree so that it can be shared and managed by multiple components. This is often used when multiple components need to share and synchronize the same piece of state.

Let's say you have a parent component that renders two child components. These child components need to share the same piece of data or state. Instead of managing this state separately within each child component, you can lift the state up to the parent component and pass it down to the children as props.

```
import React, { useState } from "react";

// Child component that displays a counter
function Counter(props) {
  return <div>Counter: {props.value}</div>;
}

// Parent component that manages the counter state and passes it to the
child components
function App() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount (count + 1);
  };

  return (
    <div>
      <Counter value={ count } />
      <button onClick={increment}>Increment</button>
    </div>
  );
}

export default App;
```

Fetch Data From API: Using Fetch/Axios

```
import React, { useState, useEffect } from "react";
//import axios from "axios";

const UsersList = () => {
  // State variable to store the list of employees
  const [users, setUsers] = useState([]);

  // Fetching data from the API using useEffect
  useEffect(() => {
    const fetchData = async () => {
      try {
        //const response = await axios.get(
        const response = await fetch(
          "https://jsonplaceholder.typicode.com/users"
        );
        const data = await response.json();
        setUsers(data);
        //setUsers(response.data);

      } catch (error) {
        console.log("Error fetching data:", error);
      }
    };

    fetchData();
  }, []);

  return (
    <div>
      {users.map((user) => (
        <div key={user.id}>
          <p>Id: {user.id}</p>
          <p>Name: {user.name}</p>
          <p>User Name: {user.username}</p>
          <p>Email: {user.email}</p>
        </div>
      ))}
    </div>
  );
};

export default UsersList;
```

Axios:

Axios is a popular JavaScript library used for making HTTP requests from the browser or Node.js. It provides an easy-to-use API for sending asynchronous HTTP requests and handling responses. You can use it to fetch data from APIs, send data to servers, and perform various HTTP operations.

Output in Table:

```
<table>
  <thead>
    <tr>
      <th>ID</th>
      <th>Name</th>
      <th>Email</th>
    </tr>
  </thead>
  <tbody>
    {data.map((user) => (
      <tr key={user.id}>
        <td>{user.id}</td>
        <td>{user.name}</td>
        <td>{user.email}</td>
      </tr>
    ))}
  </tbody>
</table>
```

Router:

A Router is a library or component that helps you manage the navigation and routing of your application. It enables you to create different views or pages and switch between them based on the URL or user interactions.

```
import React from "react";
import { BrowserRouter as Router, Route, Routes, Link } from "react-router-dom";

// Components for the routes
const Home = () => <h2>Home Page</h2>;
const About = () => <h2>About Page</h2>;
const NotFound = () => <h2>404 Not Found</h2>;

function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
        </ul>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </Router>
  );
}

export default App;
```

Controlled Component & Uncontrolled Component:

A controlled component is a component where React controls the form elements and keeps the state in sync. An uncontrolled component is a component where the form elements' state is managed by the DOM itself.

Controlled Component: The value of the input field is controlled by the state of the component (`inputValue`). Every change in the input updates the state, and the state is the single source of truth.

Uncontrolled Component: The value of the input field is controlled by the DOM itself, and you access the value using a ref (`inputRef`).

```
import React, { useState } from "react";
//import React, { useRef } from "react";

function App() {
  //const inputRef = useRef(null);
  const [inputValue, setInputValue] = useState("");

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    //alert("Submitted value: " + inputRef.current.value);
    alert("Submitted value: " + inputValue);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
```

```
        {/*Uncontrolled Input:
        <input type="text" ref={inputRef} /><br/><br /> */}
        Controlled Input:
        <input type="text" value={inputValue} onChange={handleChange}
/><br/><br />
      </label>
      <button type="submit">Submit</button>
    </form>
  );
}

export default App;
```

Display Text and Check Box:

```
import React, { useState } from "react";

const Form = () => {
  const [name, setName] = useState("");
  const [isChecked, setIsChecked] = useState(false);

  const handleNameChange = (event) => {
    setName(event.target.value);
  };

  const handleCheckboxChange = (event) => {
    setIsChecked(event.target.checked);
  };

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("Name:", name);
    console.log("Checkbox:", isChecked);
  };

  return (
    <form onSubmit={handleSubmit}>
      <div>
```

```
        <label>
          Name:
          <input type="text" value={name} onChange={handleNameChange} />
        </label>
      </div>
      <div>
        <label>
          <input
            type="checkbox"
            checked={isChecked}
            onChange={handleCheckboxChange}
          />
          Checkbox
        </label>
      </div>
      <button type="submit">Submit</button>
    </form>
  );
};
export default Form;
```

Change Button Color:

```
import React, { useState } from "react";
function App() {
  const [color, setColor] = useState("crimson");
  const changecolor = () => {
    setColor("blue");
  }
  return (
    <div>
      <button style={{ background: color }} onClick={changecolor}>
        Click here
      </button>
    </div>
  );
}

export default App;
```

React program that randomly changes the background color / 2 colors of the page when a button is clicked:

```
import React, { useState } from "react";

function App() {
  const [backgroundColor, setBackgroundColor] = useState("white"); //
Initial background color is white

  const changetoBlue = () => {
    // Change background color to blue
    setBackgroundColor("blue");
  };

  const changetoGreen = () => {
    // Change background color to green
    setBackgroundColor("green");
  };
  const changeRandomColor = () => {
    // Generate a random color
    const randomColor = "#" + Math.floor(Math.random() *
16777215).toString(16);
    setBackgroundColor(randomColor); // Fix: Use setBackgroundColor to
update state
  };

  return (
    <div style={{ backgroundColor, height: "200px" }}>
      <button onClick={changetoBlue}>Change to Blue</button>
      <button onClick={changetoGreen}>Change to Green</button>
      <button onClick={changeRandomColor}>Change Background Color</button>
    </div>
  );
}
```

```
export default App;
```

Generate a table, the cells clickable and changes the bg color to red while we click any cell.

```
import React, { useState } from "react";

const DynamicTable = () => {
  const [rows, setRows] = useState(3);
  const [columns, setColumns] = useState(3);
  const [lastClickedCell, setLastClickedCell] = useState(null);

  const generateTable = () => {
    const table = [];
    let counter = 1;

    for (let i = 0; i < rows; i++) {
      const row = [];
      for (let j = 0; j < columns; j++) {
        row.push(counter++);
      }
      table.push(row);
    }
    return table;
  };

  const cellClickHandler = (cellValue) => {
    setLastClickedCell(cellValue);
  };

  const isCellClicked = (cellValue) => {
    return lastClickedCell === cellValue;
  };

  return (
    <div>
      <label htmlFor="rows">Rows: </label>
      <input
        type="number"
        id="rows"
        value={rows}
        onChange={(e) => setRows(parseInt(e.target.value, 10))}
      />

      <label htmlFor="columns">Columns: </label>
      <input
        type="number"
        id="columns"
        value={columns}
        onChange={(e) => setColumns(parseInt(e.target.value, 10))}
      />

      <button onClick={generateTable}>Generate Table</button>

      <table style={{ borderCollapse: "collapse", margin: "20px" }}>
        <tbody>
          {generateTable().map((row, rowIndex) => (
            <tr key={rowIndex}>
              {row.map((cell, cellIndex) => (
                <td
                  key={cellIndex}
                  style={{
                    border: "1px solid #ddd",
                    width: "50px",
                    height: "50px",
                    cursor: "pointer",
                    backgroundColor: isCellClicked(cell) ? "red" : "white"
                  }}
                  onClick={() => cellClickHandler(cell)}
                >
                  {cell}
                </td>
              ))}
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
};

export default DynamicTable;
```

Payment options:

```
import React, { useState } from "react";

const PaymentOptions = () => {
  const [selectedOption, setSelectedOption] = useState(""); //
'card' or 'upi'
  const [amount, setAmount] = useState("");
  const [upiId, setUpiId] = useState("");

  const handleOptionChange = (option) => {
    setSelectedOption(option);
  };

  const handlePaymentSubmit = (e) => {
    e.preventDefault();

    // Add validation logic
    if (
      amount &&
      ((selectedOption === "card" && /* validate card details */
true) ||
        (selectedOption === "upi" && upiId))
    ) {
      // Add logic for handling the payment submission based on the
selected option, amount, and UPI ID
      console.log(
        `Payment submitted via ${selectedOption} for amount
${amount} with UPI ID ${upiId}:`
      );

      // Display success alert
      window.alert("Payment successful!");
    } else {
      // Display error alert if validation fails
      window.alert("Invalid input. Please check your details and try
again.");
```

```
    }
  };

  return (
    <div>
      <h2>Select Payment Option</h2>
      <form onSubmit={handlePaymentSubmit}>
        <label>
          Enter Amount:
          <input
            type="number"
            value={amount}
            onChange={(e) => setAmount(e.target.value)}
            required
          />
        </label>
        <br />

        <label>
          <input
            type="radio"
            value="card"
            checked={selectedOption === "card"}
            onChange={() => handleOptionChange("card")}
          />
          Credit Card
        </label>
        <label>
          <input
            type="radio"
            value="upi"
            checked={selectedOption === "upi"}
            onChange={() => handleOptionChange("upi")}
          />
          UPI
        </label>

        {selectedOption === "card" && (
          <div>
            <h3>Credit Card Details</h3>
            {/* Include credit card form component here */}
          </div>
        )}

        {selectedOption === "upi" && (
          <div>
            <h3>UPI Details</h3>
            <label>
              Enter UPI ID:
              <input
                type="text"
                value={upiId}
                onChange={(e) => setUpiId(e.target.value)}
                required
              />
            </label>
          </div>
        )}

        <button type="submit">Submit Payment</button>
      </form>
    </div>
  );
};

export default PaymentOptions;
```

React Phone Book application:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

const style = {
  table: {
    borderCollapse: 'collapse'
  },
  tableCell: {
    border: '1px solid gray',
    margin: 0,
    padding: '5px 10px',
    width: 'max-content',
    minWidth: '150px'
  },
  form: {
    container: {
      padding: '20px',
      border: '1px solid #F0F8FF',
      borderRadius: '15px',
      width: 'max-content',
      marginBottom: '40px'
    },
    inputs: {
      marginBottom: '5px'
    },
    submitBtn: {
      marginTop: '10px',
      padding: '10px 15px',
      border: 'none',
      backgroundColor: 'lightseagreen',
      fontSize: '14px',
      borderRadius: '5px',
      cursor: 'pointer'
    }
  }
}

function PhoneBookForm({ addEntryToPhoneBook }) {
  const [userFirstname, setUserFirstname] = useState('');
  const [userLastname, setUserLastname] = useState('');
  const [userPhone, setUserPhone] = useState('');

  const handleFormSubmit = (e) => {
    e.preventDefault();
    // Pass the entered information to the parent component
    addEntryToPhoneBook({
      userFirstname,
      userLastname,
      userPhone
    });
    // Clear the form after submitting
    setUserFirstname('');
    setUserLastname('');
    setUserPhone('');
  };

  return (
    <form onSubmit={handleFormSubmit} style={style.form.container}>
      <label>First name:</label>
      <br />
      <input
        style={style.form.inputs}
        className='userFirstname'
        name='userFirstname'
        type='text'
        value={userFirstname}
        onChange={(e) => setUserFirstname(e.target.value)}
      />
      <br/>
      <label>Last name:</label>
      <br />
      <input
        style={style.form.inputs}
        className='userLastname'
        name='userLastname'
        type='text'
        value={userLastname}
        onChange={(e) => setUserLastname(e.target.value)}
      />
      <br />
      <label>Phone:</label>
      <br />
      <input
        style={style.form.inputs}
        className='userPhone'
        name='userPhone'
        type='text'
        value={userPhone}
        onChange={(e) => setUserPhone(e.target.value)}
      />
      <br/>
      <input
        style={style.form.submitBtn}
        className='submitButton'
        type='submit'
        value='Add User'
      />
    </form>
  );
}

function InformationTable({ entries }) {
  return (
    <table style={style.table} className='informationTable'>
      <thead>
        <tr>
          <th style={style.tableCell}>First name</th>
          <th style={style.tableCell}>Last name</th>
          <th style={style.tableCell}>Phone</th>
        </tr>
      </thead>
      <tbody>
        {entries.map((entry, index) => (
          <tr key={index}>
            <td style={style.tableCell}>{entry.userFirstname}</td>
            <td style={style.tableCell}>{entry.userLastname}</td>
            <td style={style.tableCell}>{entry.userPhone}</td>
          </tr>
        ))}
      </tbody>
    </table>
  );
}

function Application() {
  const [entries, setEntries] = useState([]);
  const addEntryToPhoneBook = (entry) => {
    // Update the state with the new entry
    setEntries([...entries, entry]);
  };

  return (
    <section>
      {/* Pass the addEntryToPhoneBook function to the PhoneBookForm
*/}
      <PhoneBookForm addEntryToPhoneBook={addEntryToPhoneBook} />
      {/* Pass the entries to the InformationTable */}
      <InformationTable entries={entries} />
    </section>
  );
}

const container = document.getElementById('root');
const root = createRoot(container);
root.render(<Application />);
```

React Quiz App:

```
import React, { useState } from 'react';
import { createRoot } from 'react-dom/client';

const style = {
  container: {
    padding: '20px',
    border: '1px solid #E0E0E0',
    borderRadius: '15px',
    width: 'max-content',
    marginBottom: '40px',
  },
  question: {
    fontWeight: 'bold',
    marginBottom: '10px',
  },
  options: {
    marginBottom: '5px',
  },
  button: {
    marginTop: '10px',
    padding: '10px 15px',
    border: 'none',
    backgroundColor: '#007BFF',
    color: '#FFF',
    fontSize: '14px',
    borderRadius: '5px',
    cursor: 'pointer',
  },
  feedback: {
    marginTop: '10px',
    fontSize: '14px',
  },
};

// Define the questions array before the component function
const questions = [
  {
    question: 'What is the capital of France?',
    options: ['London', 'Paris', 'Berlin', 'Madrid'],
    correct: 'Paris',
  },
  {
    question: 'What is the capital of Germany?',
    options: ['Berlin', 'Munich', 'Frankfurt', 'Hamburg'],
    correct: 'Berlin',
  },
];

function QuizApp() {
  const [currentQuestion, setCurrentQuestion] = useState(0);
  const [userAnswers, setUserAnswers] =
useState(Array(questions.length).fill(''));
  const [feedback, setFeedback] = useState('');

  const handleOptionChange = (event) => {
    const selectedOption = event.target.value;
    const updatedUserAnswers = [...userAnswers];
    updatedUserAnswers[currentQuestion] = selectedOption;
    setUserAnswers(updatedUserAnswers);
  };

  const handleQuizSubmit = () => {
    const currentAnswer = userAnswers[currentQuestion];
    const correctAnswer = questions[currentQuestion].correct;

    setFeedback(currentAnswer === correctAnswer ? 'Correct!' :
'Incorrect!');
    setCurrentQuestion(currentQuestion + 1);
  };

  return (
    <div style={style.container}>
      {currentQuestion < questions.length ? (
        <div>
          <div id="question" style={style.question}>
            {questions[currentQuestion].question}
          </div>
          <div style={style.options}>
            {questions[currentQuestion].options.map((option, index) =>
(
              <div key={index} className="option">
                <input
                  type="radio"
                  id={`option${index + 1}`}
                  name="quizOptions"
                  value={option}
                  checked={userAnswers[currentQuestion] === option}
                  onChange={handleOptionChange}
                />
                <label htmlFor={`option${index + 1}`}>{option}</label>
              </div>
            ))}
          </div>
          <button style={style.button} id="submitBtn"
onClick={handleQuizSubmit}>
            Submit
          </button>
          {feedback && <div id="feedback"
style={style.feedback}>{feedback}</div>}
        </div>
      ) : (
        <div>
          <p id="quizComplete" style={style.feedback}>
            Quiz Complete! You have finished all questions.
          </p>
        </div>
      )}
    </div>
  );
}

const container = document.getElementById('root');
const root = createRoot(container);
root.render(<QuizApp />);
```

Display a calendar where Saturdays and Sundays are highlighted in red:

```
import React, { useState } from "react";
import { format, startOfMonth, addDays, isSaturday, isSunday } from
"date-fns";

const Calendar = () => {
  const [currentMonth, setCurrentMonth] = useState(new Date());

  const getMonthDays = () => {
    const startOfMonthDate = startOfMonth(currentMonth);
    const days = [];

    for (let i = 0; i < 31; i++) {
      const currentDate = addDays(startOfMonthDate, i);
      days.push(currentDate);
    }
    return days;
  };

  const renderCalendar = () => {
    const days = getMonthDays();

    return days.map((day) => (
      <div
        key={day.toISOString()}
        style={{
          padding: "10px",
          border: "1px solid #ccc",
          backgroundColor: isSaturday(day) || isSunday(day) ? "red" :
"white",
          color: isSaturday(day) || isSunday(day) ? "white" : "black",
        }}
      >
        {format(day, "d")}
      </div>
    ));
  };

  const prevMonth = () => {
    setCurrentMonth((prevMonth) => addDays(startOfMonth(prevMonth), -
1));
  };

  const nextMonth = () => {
    setCurrentMonth((prevMonth) => addDays(startOfMonth(prevMonth),
35));
  };

  return (
    <div
      style={{
        fontFamily: "Arial, sans-serif",
        textAlign: "center",
        width: "300px",
        margin: "20px auto",
      }}
    >
      <h2>{format(currentMonth, "MMMM yyyy")}</h2>
      <div
        style={{
          display: "flex",
          justifyContent: "space-between",
          marginBottom: "10px",
        }}
      >
        <button onClick={prevMonth}>&lt;</button>
        <button onClick={nextMonth}>&gt;</button>
      </div>
      <div
        style={{
          display: "grid",
          gridTemplateColumns: "repeat(7, 1fr)",
          gap: "5px",
        }}
      >
        {renderCalendar()}
      </div>
    </div>
  );
};

export default Calendar;
```