Class-07: State Management

1. Introduction to State Management

State management is a crucial aspect of building scalable and maintainable React applications. It refers to handling and sharing application state across components efficiently.

Why is State Management Important?

- Ensures consistency in data throughout the application.
- Facilitates better component reusability.
- Improves performance by preventing unnecessary re-renders.

2. Context API

The **Context API** is a built-in feature in React that provides a way to share state globally without prop drilling.

Key Concepts:

- **React.createContext()**: Creates a Context object.
- **Provider**: Wraps components and provides the state.
- **Consumer / useContext Hook**: Accesses the context value.

Example:

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

const ThemeContext = createContext();

const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light');
  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};

const ThemedComponent = () => {
  const { theme, setTheme } = useContext(ThemeContext);
  return (
```

```
    <div>
      <p>Current Theme: {theme}</p>
      <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
        Toggle Theme
      </button>
    </div>
  );
};

const App = () => (
  <ThemeProvider>
    <ThemedComponent />
  </ThemeProvider>
);

export default App;
```

Pros of Context API:

- Simple to use.
- Reduces prop drilling.
- Built into React, requiring no external dependencies.

Cons of Context API:

- Can cause unnecessary re-renders in large applications.
- Not optimized for frequent state updates.

3. React Redux (Actions, Reducers, Store)

Redux is a predictable state container for JavaScript applications that helps manage complex state changes efficiently.

Core Concepts:

1. **Actions** - Plain JavaScript objects describing what should happen.
2. **Reducers** - Functions that determine how the state changes based on actions.
3. **Store** - A centralized state container that holds the entire application state.
4. **Dispatch** - Sends an action to the store to trigger a state change.
5. **Selectors** - Functions to extract data from the store.

Steps to Implement Redux:

1. Install Redux and React-Redux:

```
npm install redux react-redux
```

2. Define Actions:

```
// actions.js
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';

export const increment = () => ({ type: INCREMENT });
export const decrement = () => ({ type: DECREMENT });
```

3. Create Reducer:

```
// reducer.js
import { INCREMENT, DECREMENT } from './actions';

const initialState = { count: 0 };

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

export default counterReducer;
```

4. Create Store:

```
// store.js
import { createStore } from 'redux';
import counterReducer from './reducer';

const store = createStore(counterReducer);

export default store;
```

5. Provide Store to React Application:

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
```

```
  </Provider>,
  document.getElementById('root')
);
```

6. Use Redux State in Components:

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';

const Counter = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h2>Counter: {count}</h2>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
};

export default Counter;
```

Pros of Redux:

- Centralized state management.
- Predictable state updates.
- Easier debugging with Redux DevTools.

Cons of Redux:

- Requires boilerplate code.
- May be overkill for small applications.

4. Middleware (Redux Thunk)

Middleware in Redux allows us to handle asynchronous operations before they reach the reducer. **Redux Thunk** is a popular middleware that allows action creators to return functions instead of plain objects.

Install Redux Thunk:

```
npm install redux-thunk
```

Apply Middleware to Store:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import counterReducer from './reducer';

const store = createStore(counterReducer, applyMiddleware(thunk));
```

Example: Fetch Data Using Redux Thunk

```
// actions.js
export const FETCH_DATA_REQUEST = 'FETCH_DATA_REQUEST';
export const FETCH_DATA_SUCCESS = 'FETCH_DATA_SUCCESS';
export const FETCH_DATA_FAILURE = 'FETCH_DATA_FAILURE';

export const fetchData = () => {
  return async (dispatch) => {
    dispatch({ type: FETCH_DATA_REQUEST });
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      const data = await response.json();
      dispatch({ type: FETCH_DATA_SUCCESS, payload: data });
    } catch (error) {
      dispatch({ type: FETCH_DATA_FAILURE, error: error.message });
    }
  };
};
```

Benefits of Redux Thunk:

- Allows handling of async API calls.
- Reduces complexity in reducers.
- Enables better control over state updates.

5. Conclusion

- **Context API** is suitable for small to medium applications.
- **Redux** is ideal for large applications with complex state.
- **Middleware like Redux Thunk** helps manage async operations efficiently.

Understanding these state management techniques helps in building robust and scalable React applications!