upGrad ENTERPRISE

State Management in React JS

By
Narasimha Rao T
Microsoft.Net FSD Trainer
Professional Development Trainer
tnrao.trainer@gmail.com

What is Prop Drilling in React?

1. What is Prop Drilling in React?

- Prop drilling refers to the process of passing data (props) through multiple levels of nested components in a React application, even when intermediate components do not directly use that data.
- This creates unnecessary coupling and can make the codebase harder to maintain as the component tree grows deeper.

Key Characteristics:

- **Unavoidable in Small Apps**: In simple hierarchies, it's fine, but it becomes problematic in large-scale apps.
- **Issues Caused**:
  - Increases boilerplate code (repetitive prop passing).
  - Makes refactoring difficult (changing a prop name requires updates across all levels).
  - Reduces component reusability.
  - Can lead to performance issues if props are complex objects.

Example Scenario:

Imagine a UserProfile component that needs user data from the top-level App component, but it's buried 5 levels deep:

```
App → Header → Navigation → Sidebar → UserProfile
```

You'd pass `user` prop from App all the way down, even if Header, Navigation, etc., don't use it.

2. Different Solutions to Address Prop Drilling

To avoid prop drilling, React provides built-in and third-party tools for global state management. Common solutions include:

- **React Context API**: Built-in way to share state across the component tree without manual prop passing. (Ideal for medium-sized apps.)
- **State Management Libraries**:
  - **Redux**: Centralized store for predictable state management. (Best for complex apps with many interactions.)
  - **Zustand**: Lightweight alternatives to Redux for simpler needs.

Choose based on app complexity: Context for simple sharing, Redux for advanced logic.

3. Context API

Introduction

React Context API is a built-in feature (introduced in React 16.3) that allows you to share state across the entire component tree without prop drilling. It creates a "context" that components can subscribe to via a Provider and consume via a Consumer or the `useContext` hook.

- **Use Cases**: Theme switching, user authentication, localization—any data needed by many components.
- **Pros**: No external dependencies; simple for medium apps.
- **Cons**: Can cause unnecessary re-renders if not optimized; not ideal for highly complex state logic.

Steps to Implement Context API

1. **Create a Context**: Use `React.createContext()` to define the context object.
2. **Provide the Value**: Wrap your app (or subtree) in a `<Context.Provider value={...}>` and pass the state/value as `value`.
3. **Consume the Context**:
   - **Class Components**: Use `<Context.Consumer>` with a render prop.
   - **Functional Components**: Use the `useContext(Context)` hook.
4. **Update State**: Manage state inside the Provider (e.g., via `useState` or `useReducer`).
5. **Optimize**: Split contexts for different data types to avoid re-renders.

4. Redux Introduction

Redux is a predictable state container for JavaScript apps, often used with React. It enforces a unidirectional data flow: State is read-only, changes happen via pure functions (reducers), and actions describe "what happened."

- **Core Principles**:
  - Single source of truth (one store).
  - State is read-only (dispatch actions to change).
  - Changes via pure reducers.

- **Use Cases**: Large apps with complex state interactions (e.g., e-commerce carts, real-time dashboards).
- **Pros**: Predictable, debuggable (time-travel debugging), middleware support (e.g., Redux Thunk for async).
- **Cons**: Boilerplate-heavy; overkill for simple apps.

Redux vs Context API

Aspect	Context API	Redux
Setup Complexity	Simple (built-in, no install)	More boilerplate (install `@reduxjs/toolkit`)
Scalability	Good for small-medium apps	Excellent for large, complex state logic
Performance	Can cause re-renders on value changes	Optimized with selectors (e.g., `useSelector`)
When to Use	Simple shared state (themes, auth)	Global state with actions/reducers

Rule of Thumb: Use Context for quick sharing; Redux for apps needing strict patterns and middleware.

5. Key Players in Redux

Store: Details

The Store is the single JavaScript object that holds the entire application state. It's created once and acts as the "single source of truth."

- **Responsibilities**:
  - Holds the state tree.
  - Allows access to state via `getState()`.
  - Dispatches actions via `dispatch(action)`.
  - Subscribes to changes via `subscribe(listener)`.
- **Creation**: Use `configureStore()` from `@reduxjs/toolkit` (modern way).

- **Key Methods**:
  - `store.getState()`: Returns current state.
  - `store.dispatch(action)`: Triggers reducers.
- **One per App**: Never create multiple stores.

Actions: Details

Actions are plain JavaScript objects that describe "what happened" in the app. They are the only way to trigger state changes.

- **Structure**: `{ type: string, payload: any }` (type is required; payload is optional data).
- **Creation**: Use action creators (functions returning actions) for reusability.
- **Types**: Synchronous (simple objects) or async (via middleware).
- **Example**:

```
const addTodo = (text) => ({
  type: 'todos/add',
  payload: { id: Date.now(), text }
});
```

- **Dispatching**: `dispatch(addTodo('Learn Redux'))`.

Reducers: Details

Reducers are pure functions that specify how the state changes in response to an action. They take current state and action, returning a new state (never mutate!).

- **Signature**: `(state, action) => newState`.
- **Rules**:
  - Pure: Same inputs → same output; no side effects.
  - Immutable: Use spread operators.
  - Initial State: Provide default if `state` is `undefined`.

- Example:

```
const todosReducer = (state = [], action) => {
  switch (action.type) {
    case 'todos/add':
      return [...state, action.payload];
    case 'todos/remove':
      return state.filter(todo => todo.id !== action.payload);
    default:
      return state;
  }
};
```

6. Steps to Implement Redux to Manage State in an Application

Using `@reduxjs/toolkit` (recommended for simplicity):

1. **Install Dependencies**: `npm install @reduxjs/toolkit react-redux`.
2. **Create Reducer**: Create reducers and actions
3. **Configure Store**: Use `configureStore()` to set up the store with reducers and middleware.
4. **Provide Store**: Wrap app in `<Provider store={store}>` from `react-redux`.
5. **Connect Components**:
    - Read state: `useSelector((state) => state.todos)`.
    - Dispatch actions: `useDispatch()`.

upGrad ENTERPRISE

Q & A

Narasimha Rao T
tnrao.trainer@gmail.com