React State Management with Redux

Slide 1: Welcome & Agenda

Welcome to the Redux Workshop!

Why state management is critical:

- Local state works for simple apps
- Global state is key for scaling

Agenda:

1. Recap: Local State & Prop Drilling
2. Global State solutions
3. Introduction to Redux concepts
4. Hands-on: Counter App and To-Do List
5. Stretch goal: Shopping Cart

Slide 2: Recap – Local State with `useState`

What is Local State?

- Managed with React's `useState`
- Works for self-contained components

Example:

```
const [count, setCount] = useState(0);
```

Limitation:

- Local state is insufficient for sharing data across components

Slide 3: The Problem – Prop Drilling

What is Prop Drilling?

- Passing state down through multiple components via props

Challenges:

- Components become tightly coupled
- Difficult to maintain

Example Code:

```
function Parent() {
  const [isModalOpen, setIsModalOpen] = useState(false);
  return <Child toggleModal={() => setIsModalOpen(!isModalOpen)} />;
}
```

Slide 4: Why Global State?

What is Global State?

- Centralized state shared across multiple components

When to Use:

- State is deeply nested
- Examples:
  - User authentication
  - Cart items

Solutions:

- Context API
- State management libraries (e.g., Redux)

Slide 5: Brief Intro to Context API

How Context API Works

1. Create a context:
   ```
   const ThemeContext = React.createContext();
   ```
2. Wrap with `Provider`:
   ```
   <ThemeContext.Provider value="dark">
   ```
3. Access with `useContext`:
   ```
   const theme = useContext(ThemeContext);
   ```

Limitations:

- Re-renders for all components
- Complex setups in large apps

Slide 6: Why Redux?

Challenges with Context API:

- Not ideal for large-scale apps
- Limited performance in deeply nested trees

Benefits of Redux:

- Predictable state transitions
- Centralized state
- Great for debugging with Redux DevTools

Slide 7: Core Concepts of Redux

Key Elements:

1. **Store**: Global state container
2. **Actions**: Objects describing events
3. **Reducers**: Pure functions to update state
4. **Dispatch**: Sends actions to the store

Slide 8: Redux Workflow

Data Flow:

1. **Dispatch** an action
2. **Action** goes to the reducer
3. **Reducer** updates the store
4. Store **notifies UI components**

Slide 9: Setting Up Redux

1. Install:

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

2. Create a store:

   ```
   const store = createStore(reducer);
   ```

3. Connect React with `Provider`:

   ```
   <Provider store={store}>
   ```

Slide 10: Building a Counter App

Features:

- Increment, decrement, reset

Example Reducer:

```
const initialState = { count: 0 };

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

Slide 11: Why Redux Toolkit?

Simplifying Redux:

- Combines actions and reducers with `createSlice`
- Easier store setup with `configureStore`

Slide 12: Setting Up Redux Toolkit

1. Install:

```
npm install @reduxjs/toolkit
```

2. Create a slice:

```
const counterSlice = createSlice({
  name: "counter",
  initialState: { count: 0 },
  reducers: { increment, decrement, reset },
});
```

3. Use `configureStore`:

```
const store = configureStore({
  reducer: { counter: counterSlice.reducer },
});
```

Slide 13: Connecting Redux Toolkit to React

1. Wrap with `Provider`:

   ```
   <Provider store={store}>
   ```

2. Use `useSelector` to read state
3. Use `useDispatch` to send actions

Slide 14: Hands-On: Counter App

Features:

- Increment, decrement, reset

Counter Component:

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

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => dispatch(increment())}>+</button>
      <button onClick={() => dispatch(decrement())}>-</button>
    </div>
  );
}
```

Slide 15: To-Do List App – Overview

Features:

- Add tasks
- Toggle completion
- Delete tasks

Slide 16: To-Do List Slice

Reducers:

- **Add Task**:

  ```
  state.todos.push({ id: Date.now(), text, completed: false });
  ```

- **Toggle Task**:

  ```
  const todo = state.todos.find((t) => t.id === action.payload);
  todo.completed = !todo.completed;
  ```

Slide 17: Building the To-Do UI

Display Tasks:

```
todos.map(todo => (
  <li>{todo.text}</li>
));
```

Dispatch Actions:

```
dispatch(addTodo("Task 1"));
dispatch(toggleTodo(id));
dispatch(deleteTodo(id));
```

Slide 18: Shopping Cart App

Features:

- Add items
- Update quantities
- Calculate totals

Slide 19: Shopping Cart Slice

Reducers:

- **Add Item**:

```
const item = state.items.find(i => i.id === action.payload.id);
if (item) { item.quantity += 1; }
```

- **Remove Item**:

```
state.items = state.items.filter(i => i.id !== action.payload);
```

Slide 20: Wrap-Up

Key Takeaways:

- Redux Toolkit simplifies global state
- Easy setup for slices and store
- Modular, clean, and scalable

Next Steps

- Practice these concepts
- Build small projects
- Explore advanced Redux patterns