Session 3 Instructor Guide: Shared State with Context

Learning Outcomes

By the end of Session 3, students will be able to:

1. **Explain** what state is in React and how it enables dynamic, interactive components
2. **Compare** state and props to understand how data flows and changes in a React app
3. **Distinguish** between local and shared state and choose the right type for different scenarios
4. **Define hooks** as functions starting with "use" that provide React features
5. **Use** built-in React hooks like useState and useContext to manage local and shared state
6. **Navigate Context pattern** and explain how custom hooks access shared state
7. **Trigger** state changes through user interactions and event handlers
8. **Use constants** to organize code and prevent errors
9. **Inspect** state changes using React DevTools for real-time debugging
10. **Create and manage** local state with useState for component-specific behavior
11. **Access shared state** through custom hooks and Context API

Instruction

Instructor introduces key concepts students need to succeed:

1. **Introducing State** - Define state as component memory that can change and trigger re-renders
2. **State vs Props** - Compare state (internal, changeable) with props (external, read-only)
3. **Local vs Shared State** - Distinguish component-specific from app-wide state with examples
4. **React Hooks** - Define hooks as "use" functions that provide React features
5. **Context API** - Explain how Context provides shared state without prop drilling
6. **Constants and Conditional Rendering** - Show SCREENS constants and `&&` operator patterns

7. **Event Handlers** - Connect user interactions to state changes
8. **React DevTools** - Demonstrate state inspection and manipulation
9. **Game Flow Architecture** - Walk through screen navigation system
10. **Best Practices** - Introduce scalable state management approaches
11. **Let's Navigate!** - Kick off hands-on mission: screen navigation and modal state

---

Slide Deck Outline

Slide 1: Shared State with Context 🧠

- **Title:** "Session 3: Shared State with Context — Managing Game Flow"
- **Session 2 Recap:** "Last time: You built game components with props, styling, and click handlers"
- **Hook:** "Your app's been static — today it starts reacting."
- **Today's Mission:**
  - **Understand** the difference between state and props
  - **Implement** screen navigation with shared state
  - **Experience** React's Context API in action
  - **Add** local state for modal functionality
  - **Master** React DevTools for state inspection
- **Visual:** Game screen flow diagram showing SPLASH → PLAYING transition
- **Connection:** "From static components to dynamic, interactive navigation!"

Slide 2: State vs Props - The Data Flow Foundation 📊

- **Title:** "Understanding React's Data Management"
- **Visual:** Split-screen comparison with arrows showing data flow

Props	State
Flow **down** (parent → child)	Live **inside** components

Read-only (immutable)	**Changeable** (mutable)
Like function parameters	Like component memory
External data	Internal data

- **Analogy:** “Props are like ingredients you receive (can’t change them), State is like your kitchen’s current condition (you control it)”
- **Key Insight:** “Props communicate between components, State manages component behavior”
- **Student Preview:** “You’ll use both today — shared state for navigation, local state for modals”

Slide 3: Local vs Shared State - Choosing the Right Tool 🎯

- **Title:** “When to Use Which Type of State”
- **Visual:** Component tree showing local state (room switches) vs shared state (building power grid)

Local State (useState): - **Scope:** Single component only - **Examples:** Modal visibility, form inputs, toggle states - **Rule:** “If only one component cares, use local state”

Shared State (Context): - **Scope:** Multiple components across the app - **Examples:** User authentication, theme, current screen - **Rule:** “If multiple components need it, use shared state”

- **Today’s Examples:**
  - **Shared:** `screen` state (affects entire app navigation)
  - **Local:** `showCredits` state (only affects SplashScreen modal)
- **Key Insight:** “Choosing the right state type is a key React skill”

Slide 4: React Hooks - Your State Management Toolkit 🪝

- **Title:** “Hooks: Functions That Hook Into React Features”
- **Definition:** “Functions starting with ‘use’ that provide React capabilities”
- **Key Rules:**
  - Always start with “use” (useState, useContext, useEffect)
  - Only call at the top level of components

- Can't be called inside loops or conditions

- **useState Syntax Breakdown:**

```
const [showCredits, setShowCredits] = useState(false);
//     ^current value  ^setter function    ^initial value
```

  - **Array destructuring** pulls out current value and setter function
  - **Naming convention:** `[thing, setThing]` pattern
  - **Initial value** can be any data type

- **Today's Hooks:**
  - **useState** - Adds local state to components
  - **useContext** - Accesses shared state from Context
  - **useGame** - Custom hook that wraps useContext for cleaner code
- **Live Demo Preview:** "We'll write our first useState hook together — and see it change the UI instantly"
- **Student Connection:** "Hooks are your tools for making components dynamic and interactive"

Slide 5: Context API - Shared State Without Prop Drilling 🌐

- **Title:** "How Context Solves the Prop Drilling Problem"
- **What is Prop Drilling?**
  - **Definition:** Passing data through multiple component levels, even when intermediate components don't need that data
  - **Problem:** Like having to ask your friend to ask their friend to ask their friend for something
  - **Result:** Inefficient, annoying, and hard to maintain
- **Visual:** Before/After diagram showing prop drilling vs Context

Without Context (Prop Drilling):

```
App (has screen state)
 ↓ passes screen as prop
SplashScreen (doesn't need screen, just passes it along)
 ↓ passes screen as prop
GameButton (finally uses screen)
```

With Context:

```
GameProvider (provides screen state)
 ↓ any component can access directly
GameButton (uses useGame hook to get screen)
```

- **Context Benefits:**
  - **No prop drilling** - Skip intermediate components
  - **Global access** - Any component can access shared data
  - **Clean code** - Less prop passing, more focused components
- **Context Metaphor:** "Think of Context as your game's command center — any component can radio in for information"
- **Today's Context:** GameProvider provides screen state to entire app
- **Student Preview:** "Your useGame hook accesses this shared state from anywhere"

Slide 6: Constants - Clean Code Organization 📋

- **Title:** "Why Constants Matter for Maintainable Code"
- **The Problem:** Magic strings scattered throughout code
- **Bad Example:** `if (screen === "splash")` vs `if (screen === "spalsh")` (typo!)
- **Good Example:** `if (screen === SCREENS.SPLASH)` (autocomplete + no typos)

Constants Checklist:
✅ **Prevent typos** - Autocomplete catches errors

✅ **Single source of truth** - Change once, updates everywhere

✅ **Better refactoring** - IDE can find all usages

✅ **Self-documenting** - Clear intent and available options

- **Today's Constants:** SCREENS object with SPLASH, PLAYING, GAME_OVER
- **Best Practice:** "Real apps have hundreds of constants for maintainability"

Slide 7: Conditional Rendering - Controlling What Users See 🔀

- **Title:** "Show/Hide Components Based on State"
- **Pattern:** `{condition && <Component />}`
- **How It Works:**
  - If condition is `true` → Component renders
  - If condition is `false` → Nothing renders
- **Examples:**

```
{screen === SCREENS.SPLASH && <SplashScreen />}
{screen === SCREENS.PLAYING && <GameMap />}
{showCredits && <CreditsModal />}
```

- **Visual:** State diagram showing screen transitions
- **Key Insight:** "One piece of state controls your entire app's display"
- **Conditional Rendering Gotcha:** "Remember: `false && <Component />` renders nothing — not an error!"
- **Student Connection:** "This pattern powers navigation in most React apps"

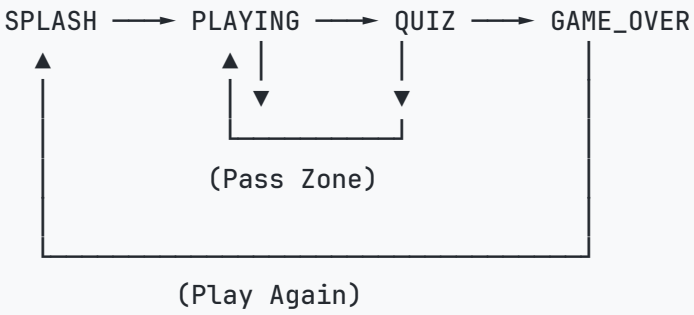
Slide 8: React DevTools - State Inspector 🔍

- **Title:** "X-Ray Vision for Your App's State"
- **Live Demo:** Show GameProvider in DevTools with state inspection
- **Key Features for State:**
  - **Component tree** - See Context relationships
  - **Hooks section** - View current state values
  - **Real-time updates** - Watch state change as you interact
  - **Manual editing** - Change state values directly for testing
- **Today's Exploration:**
  - Find GameProvider in component tree
  - Inspect screen state value
  - Manually change screen from "splash" to "playing"
  - Watch UI update instantly

- **Real-World Usage:** “Essential for debugging state-related issues”

Slide 9: Game Flow Architecture - The Big Picture 🗺️

- **Title:** “How Screen Navigation Works”
- **Visual:** State diagram showing complete game flow



- **Today’s Focus:** SPLASH ↔ PLAYING transition
- **State Control:** Single `screen` variable determines entire app display
- **Future Sessions:** Will add QUIZ and GAME_OVER screens
- **State Metaphor:** “Think of each screen as a zone in your game — and state as the teleport system”
- **Architecture Insight:** “Complex navigation is just state management”

Slide 10: Custom Hooks - Clean Code Patterns 🎨

- **Title:** “useGame: Wrapping Context for Better Developer Experience”
- **Raw Context Usage:**

```
const context = useContext(GameContext);
const { screen, setScreen } = context;
```

- **Custom Hook Usage:**

```
const { screen, setScreen } = useGame();
```

- **Benefits:**
  - **Cleaner syntax** - Less boilerplate code

    - **Error handling** - Can add validation and error messages
    - **Abstraction** - Hide implementation details
    - **Reusability** - Same hook used everywhere
- **Best Practice:** "Custom hooks are how developers organize complex state logic"

Slide 11: Build Screen Navigation! 🚀

- **Today's Implementation Journey:**
    1. **Explore** SCREENS constants for maintainable navigation
    2. **Add** screen navigation to App.jsx with conditional rendering
    3. **Use** React DevTools to inspect and manipulate state
    4. **Implement** startGame functionality in SplashScreen
    5. **Add** local state for credits modal with useState
    6. **Test** both navigation and modal functionality
- **Success Criteria:**
    - Start Adventure button navigates to GameMap
    - Credits button shows/hides modal
    - React DevTools shows state changes
- **Development Workflow:** "Build incrementally, test frequently, debug with tools"

[HANDS-ON WORK HAPPENS HERE]

Slide 12: State Management Patterns - Key Insights 💼

- **Title:** "How Real Apps Organize State"
- **Today's Patterns:**
    - **Context for global state** - App-wide data like current screen
    - **useState for local state** - Component-specific data like modal visibility
    - **Custom hooks for reusability** - Clean interfaces like useGame
    - **Constants for maintainability** - Prevent typos and improve refactoring
- **Scaling Considerations:**
    - **Small apps** - Context + useState (what you're using)
    - **Medium apps** - Add useReducer for complex state logic

    - **Large apps** - External libraries like Redux or Zustand
- **Student Empowerment:** “You’re learning patterns used in real apps”

Slide 13: What’s Next - Data-Driven Design 🏗️

- **Title:** “Preview of Session 4”
- **Today’s Achievement:** “You built navigation with shared and local state”
- **Next Challenge:** “Design your game world using data-driven architecture”
- **Concepts Coming:**
    - **Zone configuration** - Design themed trivia zones with metadata
    - **JavaScript data structures** - Arrays and objects for game content
    - **UI positioning** - Place zone labels with coordinate systems
    - **Configuration testing** - Use React DevTools to verify game scenarios
- **Motivation:** “Your GameMap will come alive with custom zones and themes!”
- **Visual:** Preview of configured game zones with positioned labels