Session 3 — Managing Game Flow

Shared State with Context 🎮

You're about to unlock one of React's most powerful features — shared state that controls your entire app! This guide walks you through implementing screen navigation, understanding the difference between local and shared state, and using React's Context API to manage game flow. Ready to make your buttons actually navigate? Let's go!

Table of Contents

- Access Your Codespace
- Understanding State vs Props
- Adding Local State for Credits
- Exploring Game Constants
- Adding Screen Navigation
- Using React DevTools to Explore State
- Implementing Start Game Functionality
- Essential Terms
- Ask the AI

☁️ Access Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 2.

🧠 Understanding State vs Props

Before we dive into code, let's understand the key difference between **state** and **props** — two fundamental concepts that control how data flows in React apps.

Props vs State: The Key Differences

Props	State
Data flows **down** from parent to child	Data lives **inside** a component

Read-only — child can't change them	**Changeable** — component can update it
Like function parameters	Like component memory
External data	Internal data

💡 Why This Matters

Props are like ingredients you receive to make a recipe — you can't change them, but you use them to create something. **State** is like your kitchen's current condition — you can rearrange, add, or remove things as needed. Understanding this difference is crucial because it determines how data flows through your app and which component is responsible for managing what information.

🏆 Adding Local State for Credits

Let's implement **local state** for the credits modal to see how components can manage their own data.

1. **Add imports** at the top of SplashScreen.jsx:

```
import { useState } from "react";
import CreditsModal from "./CreditsModal";
```

2. **Add local state** inside the SplashScreen function (before the return):

```
const [showCredits, setShowCredits] = useState(false);
```

3. **Update the Credits button**:

```
<GameButton
  text="Credits"
  onClick={() => setShowCredits(true)}
  variant="secondary"
/>
```

4. **Add the modal** before the closing `</div>` tag:

```
{showCredits && <CreditsModal onClose={() => setShowCredits(false)} />}
```

5. **Test the credits modal**: Click the Credits button to see the modal appear

💡 Why This Matters

Local state with `useState` belongs to a single component and gives it its own memory. The credits modal only affects SplashScreen, so it uses local state to track whether the modal should be visible. This pattern keeps component data isolated and manageable.

🏆 Bonus Challenge

Use React DevTools to inspect the SplashScreen component and watch the `showCredits` state change as you interact with the Credits button.

📋 Exploring Game Constants

Let's understand how our game screens are organized using **constants** — static values that prevent typos and make code more maintainable.

1. **Explore the screens constant**: Open `src/constants/screens.js` and examine the SCREENS object
2. **Notice the structure**: Each screen has a key (like `SPLASH`) and a descriptive value
3. **Understand the purpose**: Instead of using strings like `"splash"` everywhere, we use `SCREENS.SPLASH`

💡 Why This Matters

Constants prevent typos and make your code more maintainable. Instead of typing `"splash"` in multiple places (and risking typos like `"spalsh"`), you use `SCREENS.SPLASH` once and get autocomplete everywhere. If you need to change the value later, you only change it in one place.

🗺️ Adding Screen Navigation

Now let's implement the core navigation system that will control which screen users see. This is where **shared state** really shines!

1. **Open** `src/App.jsx` and add the necessary imports at the top:

```
import { useGame } from './hooks/useGame';
import { SCREENS } from "./constants/screens";
import GameMap from "./components/GameMap";
import SplashScreen from "./components/SplashScreen";
```

2. **Access the shared state** by adding this line inside the App function (before the return):

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

3. **Add conditional rendering** by replacing the current JSX with:

```
return (
  <div className="app-container">
    {screen === SCREENS.SPLASH && <SplashScreen />}
    {screen === SCREENS.PLAYING && <GameMap />}
  </div>
);
```

4. **Test the setup**: Run `npm run dev` to make sure everything still works

💡 Why This Matters

Conditional rendering using `&&` is a React pattern that shows components only when certain conditions are true. When `screen` equals `SCREENS.SPLASH`, the SplashScreen component renders. When it equals `SCREENS.PLAYING`, GameMap renders instead. This single piece of **shared state** controls what your entire app displays!

🔍 Using React DevTools to Explore State

Let's use React DevTools to see how **shared state** works behind the scenes and experiment with changing it manually.

1. **Open DevTools**: Press F12 or right-click → Inspect
2. **Find Components tab**: Look for "Components" next to Console, Network, etc.
3. **Locate GameProvider**: Click on GameProvider in the component tree
4. **Examine the hooks**: Look for the screen state value (if you don't see hook names clearly, click the gear icon and enable "Parse hook names")

5. **Experiment with state**: Change the screen value from “splash” to “playing” and watch the UI update!
6. **Change it back**: Set it back to “splash” to see the SplashScreen return

💡 Why This Matters

React DevTools gives you X-ray vision into your app’s **state**. You can see exactly what data each component has and even modify it in real-time. This is invaluable for debugging and understanding how **shared state** affects your entire app. Notice how changing one value in GameProvider instantly changes what component renders!

🏆 Bonus Challenge

Try changing the screen state to different values and see what happens. What occurs when you set it to a value that doesn’t match any of your conditions?

🎮 Implementing Start Game Functionality

Now let’s make your “Start Adventure” button actually start the game by updating the **shared state**!

1. **Open** `src/components/SplashScreen.jsx` and add imports at the top:

```
import { SCREENS } from "../constants/screens";
import { useGame } from "../hooks/useGame";
```

2. **Access the state setter** by adding this inside the SplashScreen function (before the return):

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

3. **Create the start game function** (before the return):

```
const startGame = () ⇒ {
  setScreen(SCREENS.PLAYING);
};
```

4. **Update the first GameButton** to use the real function:

```
<GameButton
  text="Start Adventure"
  onClick={startGame}
  variant="primary"
/>
```

5. **Test it**: Click the “Start Adventure” button and watch the screen change to GameMap!

💡 Why This Matters

State setters like `setScreen` are functions that update **state** and trigger re-renders. When you call `setScreen(SCREENS.PLAYING)`, React updates the shared state and re-renders all components that depend on it. This is how one button click can change your entire app’s display!

📚 Essential Terms

Quick reference for all the state management concepts you just learned:

Term	Definition	Why it matters
🧠 state	Data that can change over time and causes components to re-render when it changes.	State lets components “remember” information and respond to user interactions dynamically.
🪝 hook	Functions starting with “use” that let you use React features like state and context.	Hooks like useState and useContext are your tools for managing data and behavior in components.
🌐 Context	Lets a component receive information from distant parents without passing it as props.	Context prevents “prop drilling” and provides shared state accessible from any component.

📦 props	Data passed from parent to child components.	Props flow data down the component tree, while state manages data within components.
🔄 useState	A React hook that adds local state to functional components.	useState gives individual components their own memory for data that only they need to track.
🎯 useContext	A React hook that accesses shared data from a Context Provider.	useContext lets any component access shared state without prop drilling through multiple levels.
📋 constants	Static values that don't change, used to prevent typos and make code more maintainable.	Constants like SCREENS.SPLASH prevent typos and make refactoring easier.
🔀 conditional rendering	Showing different components based on state or props using JavaScript expressions.	Conditional rendering with && lets you control what users see based on app state.
🏪 Provider	A Context component that makes shared state available to all child components.	The Provider pattern wraps your app and gives all components access to shared data.

🤖 Ask the AI — State Management Mastery

You just implemented both local and shared state, created screen navigation, and experienced the power of React's Context API — excellent work!

Now let's deepen your understanding of state management, hooks, and the React data flow. Here are the most impactful questions to ask your AI assistant about today's session:

- **How does conditional rendering with && work in React?**
- **What makes hooks special and why do they all start with "use"?**
- **Explain** `const [showCredits, setShowCredits] = useState(false);` **in regular English.**

- **Explain state setter functions like `setScreen`, but in a non-tech example.**
- **What is “prop drilling” and how does the Context API prevent it? Give me non-tech examples.**
- **How does the GameProvider make state available to all components?**

💬 Pro Tip:

State management is the heart of React apps. Think of local state as a component’s private memory and shared state as the app’s global memory. Choose local state when only one component needs the data, and shared state when multiple components need to coordinate.