Session 8 — Implementing Scoring & Victory

Application State Management 🏆

You're about to add the most satisfying part of any game — scoring and victory! This guide walks you through implementing a complete scoring system, managing complex application state, and creating your first independent React component. Ready to make your trivia game feel like a real achievement? Let's go!

Table of Contents

☁️ Access Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 7.

🧠 Understanding Application State

Before we dive into scoring, let's understand how **application state** differs from the component state you've used before.

Application state is the complete picture of your game's current condition — everything from the player's score to which zones are completed. Think of it as your game's "save file" that tracks all progress and achievements.

State Categories in Your Game

Your GameContext manages five categories of state:

Category	Purpose	Examples
Game State	Core game progress	`score`, `screen`, `zoneProgress`
Quiz State	Current quiz session	`currentQuestions`, `currentQuestion`, `correctAnswers`
Audio	Sound controls	`music` settings
Actions	Game logic functions	`recordCorrectAnswer`, `resetGame`
Controls	UI state setters	`setScreen`, `setIsQuizVisible`

💡 Why This Matters

Application state management is what separates simple websites from complex, interactive applications. Your scoring system will coordinate multiple pieces of state to create a cohesive game experience where every action has consequences and every achievement is tracked.

🏆 Adding Score Tracking

Let's add a scoring system that tracks player performance and displays it prominently in your game's HUD.

1. **Open** `src/context/GameContext.jsx` and add score state inside the GameProvider function:

   ```
   const [score, setScore] = useState(0); // Add score state
   ```

2. **Make score available** by adding it to the Context value:

```
<GameContext value={{
  // GAME STATE
  screen,
  score, // Add score to make available
  zoneProgress,
  // ... rest of existing properties
```

3. **Open** `src/components/HUD.jsx` and add a Scoreboard component at the top of the file:

```
function Scoreboard() { // Add Scoreboard component
  const { score } = useGame();
  return <div className="score-display">Score: {score}</div>;
}
```

4. **Update HUD to render both components** using a React Fragment (`<>...</>`):

```
// Replace this single line:
return <CurrentZone />;

// With this Fragment structure:
return (
  <>
    <Scoreboard /> // Add Scoreboard
    <CurrentZone />
  </>
);
```

5. **Test**: Navigate to the game screen → Score: 0 appears in HUD

💡 Why This Matters

The **Scoreboard component** demonstrates the single responsibility principle — it has one job: display the current score. This modular approach makes your code easier to maintain and test.

📊 Implementing Score Updates

Now let's make the score actually change based on player performance with point rewards and penalties.

1. **Open** `src/context/GameContext.jsx` and find the `recordCorrectAnswer` function, then add points for correct answers:

```
const recordCorrectAnswer = () => {
  setCorrectAnswers((prev) => prev + 1);
  setScore((prev) => prev + POINTS_PER_CORRECT); // Add score increase
};
```

2. **Find the** `recordIncorrectAnswer` **function** and add point deduction:

```
const recordIncorrectAnswer = () => {
  setScore((prev) => Math.max(0, prev - POINTS_PER_CORRECT)); // Add score decre
};
```

3. **Test**: Click zone → Answer questions → Observe score changes:
   - **Correct answer** → +100 points
   - **Incorrect answer** → -100 points (but never below 0)

💡 Why This Matters

Updater functions like `setScore((prev) => prev + 100)` are crucial when updating state based on the previous value. React batches state updates, so using the previous value ensures accurate calculations even when multiple updates happen quickly.

🗄️ Adding Cache Clearing

Let's add cache clearing functions to remove stored questions when zones are completed or the game resets.

1. **Open** `src/services/trivia.js` and add cache clearing functions at the end of the file:

```
export function clearQuestionCache(zoneId) { // Add cache clearing for single zo
  const cacheKey = getCacheKey(zoneId);
  localStorage.removeItem(cacheKey);
}

export function clearAllQuestionCache() { // Add cache clearing for all zones
  Object.keys(localStorage)
    .filter((key) => key.startsWith("trivia_questions_zone_"))
    .forEach((key) => localStorage.removeItem(key));
}
```

2. **Import the cache functions** into GameContext.jsx:

```
import {
  fetchQuestions,
  clearQuestionCache,
  clearAllQuestionCache
} from "../services/trivia"; // Add cache functions
```

3. **Update the `checkZoneCompletion` function** to clear the current zone's cached questions when completed:

```
const checkZoneCompletion = () => {
  if (activeZone === null || currentQuestions.length === 0) return;

  const questionsNeeded = Math.ceil(
    currentQuestions.length * PASS_PERCENTAGE
  );
  const passed = correctAnswers >= questionsNeeded;

  if (passed) {
    setZoneProgress((prev) => ({
      ...prev,
      [activeZone]: { completed: true },
    }));

    clearQuestionCache(activeZone); // Add cache clearing

    if (activeZone === ZONES.length - 1) {
      setScreen(SCREENS.GAME_OVER);
    }
  }
};
```

💡 Why This Matters

Cache management prevents stale data from affecting gameplay. When players complete a zone, clearing its cache ensures they get fresh questions if they replay. The `Object.keys()` and `filter()` pattern is a professional way to find and remove related localStorage entries.

🔄 Updating Reset Functionality

Let's update the reset function to properly clear all game state and cached data for a fresh start.

1. **Find the `resetGame` function** in GameContext.jsx and update it:

```
const resetGame = () => {
  setScore(0); // Add score reset
  setZoneProgress({
    0: { completed: false },
    1: { completed: false },
    2: { completed: false },
  });
  setIsQuizVisible(false);
  setCurrentQuestions([]);
  setCurrentQuestion(0);
  setCorrectAnswers(0);
  clearAllQuestionCache(); // Add cache clearing
};
```

2. **Test the reset functionality** by completing a zone and then using React DevTools to trigger `resetGame()`

💡 Why This Matters

Complete state reset ensures players can start fresh without any lingering data from previous games. This includes both React state and localStorage cache, providing a clean slate for new gameplay sessions.

🎖️ Solo Mission: GameOver Component

Now for the exciting part — you'll create a GameOver component that celebrates player achievements and allows them to play again! You've got all the tools — now it's time to build your own victory screen using everything you've learned.

1. Create the Component Foundation

- Create `src/components/GameOver.jsx` with function component and default export
- Return JSX with div `className="game-over"` containing h1 congratulations message
- Import `GameOver` into `App.jsx` and add conditional rendering for `SCREENS.GAME_OVER`

- **Test**: Use React DevTools → Set `screen` to “gameover” → Component appears

2. Add Score Display

- Import `useGame` hook and destructure `score`
- Add div with `className="final-score"` displaying `Final Score: {score}`
- **Test**: Check score display → Shows current game score

3. Add Play Again Functionality

- Create click handler calling `resetGame` and `setScreen(SCREENS.SPLASH)`
- Import and render `GameButton` with “Play Again” text and `"primary"` variant
- **Test**: Click Play Again → Game resets → Returns to splash screen

Testing Tips

- **Quick testing**: Use React DevTools to change `screen` state to “gameover” (find `GameProvider` → hooks → screen)
- **Full testing**: Complete all three zones to naturally trigger GameOver screen
- **Verify**: Final score displays correctly and Play Again button resets everything

Requirements Checklist

Your completed GameOver component must:

- Export function component as default
- Wrap content in div with `className="game-over"`
- Display congratulations using h1 element
- Import `GameButton`, `useGame`, `SCREENS`
- Show final score in div with `className="final-score"`
- Include Play Again button using `GameButton` with `"primary"` variant
- Reset game and navigate to splash screen when Play Again is clicked
- Display when screen state equals `SCREENS.GAME_OVER` in `App.jsx`

🔍 Reference Files

- `SplashScreen.jsx`: Component structure, `GameButton` usage, `useGame` hook, screen navigation, click handler patterns

- `HUD.jsx`: Accessing `score` from `useGame` hook
- **Session 2 guide**: `GameButton` props and component export patterns
- **Session 3 guide**: `SCREENS` constants and navigation patterns

💡 Why This Matters

This challenge combines everything you've learned: component creation, props, shared state, event handling, and conditional rendering. Following guided steps without code examples builds confidence in applying React patterns — you're connecting concepts rather than copying code.

📚 Essential Terms

Quick reference for the key concepts you just learned:

Term	Definition	Why it matters
🏗️ application state	The complete condition of an application at a specific moment in time, encompassing all the information it needs to function correctly.	Your GameContext manages all application state — screen, score, zone progress — making it accessible to any component through useGame.
🔄 updater function	A function passed to setState that receives the previous state value and returns the new state.	Essential for score calculations — ensures accurate updates even when React batches multiple state changes.
🎯 single responsibility principle	Design pattern where each component or function has one clear, focused purpose.	Your Scoreboard component only displays score — this separation makes code easier to test and maintain.

🤖 Ask the AI — Application State Understanding

You just implemented a complete scoring system, managed complex application state, and built your first independent React component — excellent work!

Now let’s deepen your understanding of state management patterns, component architecture, and professional development practices. Here are the most impactful questions to ask your AI assistant about today’s session:

- **How does application state differ from component state, and when should I use each?**
- **Why are updater functions important for state that depends on previous values?**
- **What are some common patterns for resetting application state in React apps?**
- **What makes a component follow the single responsibility principle?**
- **How do you decide what props a new component needs when building it from scratch?**