Session 8 — Implementing Scoring & Victory

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

☁️ Accessing Your Codespace

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

🧠 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.

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`

Think of your `GameContext` as the brain of your game — it keeps track of everything that's happening behind the scenes. The `useGame` hook provides access to this brain from any component that needs it. The mind map below breaks down how your game's state is organized, with a spotlight on **actions** — these are the functions that drive your game logic and help different parts of your app work together:

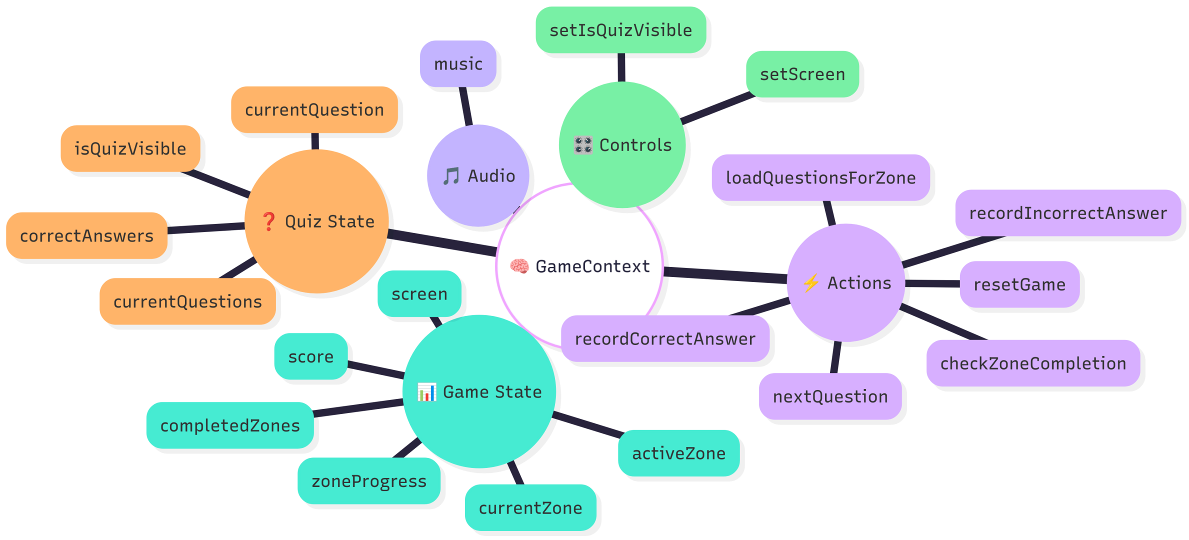

Figure: GameContext Mind Map

Managing application state 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

🎯 **Goal:** Add a scoring system that tracks player performance and displays it prominently in your game's HUD.

You'll add score state to GameContext, expose it through the Context value, create a Scoreboard component, and integrate it into the HUD.

Step 1: Add score state

File: `src/context/GameContext.jsx`

Add score state to track player performance throughout the game.

```
export function GameProvider({ children }) {
  const [screen, setScreen] = useState(SCREENS.SPLASH);
  const [score, setScore] = useState(0);  // Add score state

  // ... rest of state declarations
```

> 💡 **Understanding useState**
>
> `useState` is a React hook that lets your component remember things — like a score — between renders. It gives you:
>
> - **A current value** (`score`) that starts at `0`
> - **A setter function** (`setScore`) that can be used to update the value and trigger a re-render

Step 2: Share score with all components

File: `src/context/GameContext.jsx`

Make score available to all components by adding it to the Context value object.

```
export function GameProvider({ children }) {
  // ... state declarations ...

  return (
    <GameContext
      value={{
        // GAME STATE
        screen,
        score,  // Add score to value
        zoneProgress,
        // ... rest of properties
      }}
    >
      {children}
    </GameContext>
  );
}
```

> 💡 Understanding How State Is Shared
>
> In Step 1, you created `score` as local state.
>
> By adding it to the Context value here, you make it available across your app.
>
> Now any component that calls `useGame()` can access `score` — no prop-drilling needed.

Step 3: Create Scoreboard component

File: `src/components/HUD.jsx`

Create a new component that displays the score from shared state.

```
// Add import at top of file:
import { useGame } from "../hooks/useGame";  // [1] Import useGame hook

// Add after imports:
function Scoreboard() {
  const { score } = useGame();  // [2] Get score from shared state
  return <div className="score-display">Score: {score}</div>;  // [3] Display score
}
```

💡 Understanding Shared State Access

1. **Import useGame hook**: Access the shared game state from Context
2. **Get score from shared state**: Destructure the score value from GameContext
3. **Display score**: Render the score in a styled div

The Scoreboard component accesses the score directly from shared state using the `useGame` hook. This means any component can display or use the score without passing it through props. When the score updates in GameContext, all components using it automatically re-render with the new value.

Step 4: Add Scoreboard to HUD

File: `src/components/HUD.jsx`

Update the HUD component to render both Scoreboard and CurrentZone using a React Fragment.

```
// Before:
export default function HUD() {
  return <CurrentZone />;
}

// After:
export default function HUD() {
  return (
    <>
      <Scoreboard />
      <CurrentZone />
    </>
  );
}
```

Step 5: Test score display

Navigate to the game screen.

✓ **You should see:** “Score: 0” appears in the HUD above the current zone display.

📊 Implementing Score Updates

🎯 **Goal:** Make the score change based on player performance with point rewards and penalties.

You'll update the `recordCorrectAnswer` and `recordIncorrectAnswer` functions to modify the score when players answer questions.

Step 1: Add points for correct answers

File: `src/context/GameContext.jsx`

Update the `recordCorrectAnswer` function to award points when players answer correctly.

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

Step 2: Add point deduction for incorrect answers

File: `src/context/GameContext.jsx`

Update the `recordIncorrectAnswer` function to deduct points, preventing negative scores.

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

Step 3: Test score updates

Click a zone, then answer questions.

✓ **You should see:** - Correct answer → Score increases by 100 points - Incorrect answer → Score decreases by 100 points (but never below 0)

> 💡 Updater Functions
>
> **Updater functions** like `setScore((prev) => prev + 100)` are crucial when updating state based on the previous value. React batches state updates for performance, so using the previous value ensures accurate calculations even when multiple updates happen quickly. Without the updater function, you might lose updates or get incorrect values.

🗄️ Adding Cache Clearing

🎯 **Goal:** Add cache clearing functions to remove stored questions when zones are completed or the game resets.

You'll create functions to clear individual zone caches and all caches, then integrate them into your game flow.

Step 1: Create cache clearing functions

File: `src/services/trivia.js`

Add cache clearing functions at the end of the file.

```
export function clearQuestionCache(zoneId) {
  const cacheKey = getCacheKey(zoneId);
  localStorage.removeItem(cacheKey);
}

export function clearAllQuestionCache() {
  Object.keys(localStorage)
    .filter((key) => key.startsWith("trivia_questions_zone_"))
    .forEach((key) => localStorage.removeItem(key));
}
```

Step 2: Import cache functions into GameContext

File: `src/context/GameContext.jsx`

Add the cache clearing functions to your imports.

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

Step 3: Clear cache on zone completion

File: `src/context/GameContext.jsx`

Update the `checkZoneCompletion` function to clear the zone's cache 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);  // Clear zone cache

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

Step 4: Test cache clearing

Complete a zone, then check localStorage in browser DevTools.

✓ **You should see:** The cache entry for the completed zone is removed from localStorage.

> 💡 Cache Management
>
> 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 an effective way to find and remove related localStorage entries by matching key prefixes.

🔄 Updating Reset Functionality

🎯 **Goal:** Update the reset function to properly clear all game state and cached data for a fresh start.

You'll modify the `resetGame` function to reset all state variables and clear the question cache.

Step 1: Update resetGame function

File: `src/context/GameContext.jsx`

Update the `resetGame` function to reset all state and clear caches.

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

Step 2: Test reset functionality

Complete a zone, then use React DevTools to trigger `resetGame()`.

✓ **You should see:** All state resets to initial values and localStorage cache is cleared.

> 💡 Complete State Reset
>
> 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. Resetting all related state together prevents bugs where some state is stale while other state is fresh.

🎖️ 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** by using React DevTools → setting `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** by checking 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** by clicking Play Again → Game resets → Returns to splash screen

Testing Tips

- **Quick testing** by using React DevTools to change `screen` state to "gameover" (find `GameProvider` → hooks → screen)
- **Full testing** by completing 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

> 💡 Building Without Code Examples
>
> 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. This is how you develop the ability to build features independently.

📚 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.

🤖 Ask the AI — Implementing Scoring & Victory

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 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?**
- **How do you decide what props a new component needs when building it from scratch?**
- **What are the benefits of breaking UI into small, focused components?**