Session 7 — Creating the Quiz Experience

Building Complex Interactive Components 🎯

You're about to build the heart of your trivia game — interactive quiz components that bring your cached questions to life! This guide walks you through component composition, array mapping patterns, and creating dynamic user feedback systems. Ready to transform your static zones into engaging quiz experiences? Let's go!

Table of Contents

☁️ Access Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 6.

🧩 Understanding Component Composition

Before we start coding, let's understand how complex components are built from smaller pieces — the foundation of scalable React architecture.

Component composition is like building with LEGO blocks — you create complex structures by snapping together smaller, focused pieces. Your `QuizModal` is actually composed of five smaller components working together:

```
QuizModal (the container)
├── ProgressHeader (shows question progress)
├── QuestionHeader (displays the question)
├── AnswerChoices (interactive answer buttons)
├── AnswerFeedback (shows results)
└── ContinueButton (navigation control)
```

💡 Why This Matters

Component composition is how professional React apps stay organized and maintainable. Instead of one massive component doing everything, you break functionality into focused pieces. Each component has a single responsibility, making your code easier to understand, test, and modify.

🕹️ Connecting the Quiz Modal

Let's connect your quiz modal to the game flow so clicking zones actually shows quiz questions.

1. **Open** `src/App.jsx` and add the QuizModal import at the top:

```
import QuizModal from "./components/QuizModal"; // Add this import
```

2. **Access the quiz visibility state** by adding `isQuizVisible` to the `useGame` destructuring:

```
const { screen, isQuizVisible } = useGame(); // Add isQuizVisible
```

3. **Add conditional rendering** for the QuizModal inside the PLAYING screen section:

```
{screen === SCREENS.PLAYING && (
  <>
    <GameMap />
    <HUD />
    {isQuizVisible && <QuizModal />} // Add this line
    <CoordinateDisplay />
  </>
)}
```

   Conditional rendering with `&&` is a React pattern that shows components only when certain conditions are true. When `isQuizVisible` is true, the `QuizModal` renders; when false, nothing renders.

4. **Open** `src/components/GameMap.jsx` and add `setIsQuizVisible` to the `useGame` destructuring:

```
const { activeZone, loadQuestionsForZone, setIsQuizVisible, zoneProgress } = useGame(); // Add setIsQuizVisible
```

5. **Update the handleZoneClick function** to show the quiz modal after loading questions:

```
await loadQuestionsForZone(zoneId);
setIsQuizVisible(true); // Add this line
```

   Test: Click a zone → QuizModal should appear with your cached questions!

💡 Why This Matters

This pattern controls what users see based on app state. Your quiz modal will appear and disappear based on user actions, creating a smooth interactive experience.

🔵 Building Answer Buttons

Now let's build the interactive answer buttons that transform your question data into clickable choices.

1. **Open** `src/components/QuizModal.jsx` and find the QuestionHeader function

2. **Add the AnswerChoices component** after the QuestionHeader function:

```
function AnswerChoices({ answers }) { // Add this component
  return <div className="answers-grid"></div>;
}
```

3. **Add the component to the JSX** (right after `<QuestionHeader question={question} />`):

```
<AnswerChoices answers={question.answers} /> // Add this line
```

Test: Click zone → React DevTools → Find AnswerChoices → Confirm answers prop is populated

4. **Transform the answers array into buttons** using React's mapping pattern:

```
function AnswerChoices({ answers }) {
  return (
    <div className="answers-grid">
      {answers.map((answer, index) => ( // Turn answers into buttons
        <button key={index} className="answer-button">
          {answer}
        </button>
      ))}
    </div>
  );
}
```

Array mapping transforms each item in an array into something else. Here, each answer string becomes a button element. React needs a unique **key prop** for each mapped element to track changes efficiently.

Test: Click zone → You should see answer buttons appear in the modal

💡 Why This Matters

By starting with a simple structure and then adding the mapping logic, you followed a professional development pattern: build incrementally and test each step. **Array mapping** is everywhere in React — any time you have a list of data that becomes a list of components, you use `map()`. The **key prop** helps React optimize updates by tracking which items changed, moved, or were added/removed, making your dynamic button lists performant and reliable.

🎨 Making Buttons Interactive

Let's add click functionality and dynamic styling that shows correct/incorrect answers with visual feedback.

1. **Add click handling** by updating AnswerChoices and JSX:

```
function AnswerChoices({ answers, onAnswerClick }) { // Add onAnswerClick prop
  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (
        <button
          key={index}
          className="answer-button"
          onClick={() => onAnswerClick(index)} // Add click handler
        >
          {answer}
        </button>
      ))}
    </div>
  );
}
```

```
<AnswerChoices
  answers={question.answers}
  onAnswerClick={handleAnswerClick} // Add this prop
/>
```

Test: Click zone → Click answer button → Next Question should be enabled allowing you to move to next question

2. **Add conditional styling** by updating AnswerChoices and JSX:

```
function AnswerChoices({ answers, onAnswerClick, chosenAnswer, correctAnswer }) { // Add new props
  const getButtonStyle = (answerIndex) => { // Add styling function
    if (chosenAnswer === null) return "answer-button";
    if (answerIndex === correctAnswer) return "answer-button correct";
    if (answerIndex === chosenAnswer) return "answer-button incorrect";
    return "answer-button";
  };

  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (
        <button
          key={index}
          className={getButtonStyle(index)} // Use styling function
          onClick={() => onAnswerClick(index)}
        >
          {answer}
        </button>
      ))}
    </div>
  );
}
```

```
<AnswerChoices
  answers={question.answers}
  onAnswerClick={handleAnswerClick}
  chosenAnswer={chosenAnswer} // Add these props
  correctAnswer={question.correct}
/>
```

Test: Click zone → Click answer → Different styling for correct vs incorrect

3. **Prevent multiple clicks** by updating AnswerChoices:

```
function AnswerChoices({ answers, onAnswerClick, chosenAnswer, correctAnswer }) {
  const getButtonStyle = (answerIndex) => {
    if (chosenAnswer === null) return "answer-button";
    if (answerIndex === correctAnswer) return "answer-button correct";
    if (answerIndex === chosenAnswer) return "answer-button incorrect";
    return "answer-button";
  };

  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (
        <button
          key={index}
          className={getButtonStyle(index)}
          onClick={() => onAnswerClick(index)}
          disabled={chosenAnswer !== null} // Add disabled state
        >
          {answer}
        </button>
      ))}
    </div>
  );
}
```

Test: Click zone → Click answer → Try clicking other buttons → Other buttons should be unclickable

💡 Why This Matters

Event handlers like `onClick` connect user actions to your app's logic, making static components come alive with interactivity. **Conditional styling** then provides immediate visual feedback to users through the `getButtonStyle` function, which returns different CSS classes based on the current state. The `disabled` attribute prevents multiple clicks after an answer is selected, creating a polished user experience where buttons respond intelligently to user interactions.

💬 Adding Feedback Messages

Let's add personality to your game with custom feedback messages that celebrate correct answers and encourage players after mistakes, then make them dynamic with random selection.

1. **Create the messages file**: Right-click `src/constants` → New File → name it `messages.js`
2. **Add feedback message arrays**:

```
// Add these message arrays
export const CORRECT_FEEDBACK = [
  "🎉 Nailed it!",
  "🔥 You got it!",
  "✨ Awesome!",
  "🏆 Perfect!",
  "💯 Brilliant!",
  "⭐ Outstanding!",
  "🚀 Amazing!"
];

export const INCORRECT_FEEDBACK = [
  "😬 Missed it!",
  "💥 Not quite!",
  "🤔 Close one!",
  "😅 Try again!",
  "🎯 Almost there!",
  "💪 Keep going!",
  "🧠 Learning time!"
];
```

3. **Import the constants** into QuizModal.jsx:

```
import { CORRECT_FEEDBACK, INCORRECT_FEEDBACK } from "../constants/messages"; // Add this import
```

4. **Find the AnswerFeedback function** in QuizModal.jsx and replace the placeholder message with random selection logic:

```
function AnswerFeedback({ hasAnswered, isCorrect, correctAnswerText }) {
  if (!hasAnswered) {
    return <AnswerPlaceholder />;
  }

  const messages = isCorrect ? CORRECT_FEEDBACK : INCORRECT_FEEDBACK; // Choose message array
  const message = messages[Math.floor(Math.random() * messages.length)]; // Select random message

  return (
    <div className="result">
      <strong>{message}</strong> // Display random message
      {!isCorrect && <div>The answer was: {correctAnswerText}</div>}
    </div>
  );
}
```

5. **Add AnswerFeedback component** to JSX (right after QuestionHeader):

```
<AnswerFeedback
  hasAnswered={chosenAnswer !== null}
  isCorrect={chosenAnswer === question.correct}
  correctAnswerText={question.answers[question.correct]}
/>
```

Test: Click zone → Click answers → See different messages each time

Random Selection Breakdown

```
const messages = isCorrect ? CORRECT_FEEDBACK : INCORRECT_FEEDBACK;
// Choose the right array based on whether answer was correct

const message = messages[Math.floor(Math.random() * messages.length)];
// Math.random() → 0 to 0.999...
// * messages.length → 0 to array length
// Math.floor() → Round down to whole number
// messages[index] → Get message at that position
```

💡 Why This Matters

Constants keep your feedback messages organized and easy to modify. By storing them in a separate file, you can easily add new messages or change the tone without hunting through component code. **Random selection** adds variety and personality to your game using `Math.random()` and `Math.floor()` — the same pattern used in games, animations, and any app that needs variety.

🏆 Bonus Challenge

Add more feedback messages to each array to increase variety! Try different emojis and encouraging phrases.

🧪 Testing Your Quiz System

Let's test your complete quiz system and verify all the interactive pieces work together.

Complete Quiz Flow Test

- **Navigate to game**: Click "Start Adventure"
- **Click a zone**: Modal should appear with question and answers
- **Click an answer**:
  - Button should show correct/incorrect styling
  - Random feedback message should appear
  - Other buttons should be disabled
  - Continue button should become enabled
- **Click Continue**: Next question should load
- **Complete all questions**: Modal should close and zone should be marked complete

React DevTools Inspection

- **Open DevTools**: Press F12 → Components tab
- **Find QuizModal**: Examine the component tree

- **Inspect AnswerChoices**: Check answers prop and chosenAnswer state
- **Watch state changes**: Click answers and observe chosenAnswer updates

💡 Why This Matters

End-to-end testing ensures all your components work together correctly. By testing the complete user flow, you catch integration issues that might not appear when testing individual components.

📚 Essential Terms

Quick reference for all the component composition and interaction concepts you just learned:

Term	Definition	Why it matters
🧩 component composition	Building complex components by combining smaller, focused components together.	Your `QuizModal` is composed of five smaller components, making it easier to understand and maintain.
🗺️ Array.map()	JavaScript method that transforms each item in an array into something else, returning a new array.	Essential for converting your answers array into JSX button elements in React.
🔑 key prop	Unique identifier React needs for each element in a mapped array to track changes efficiently.	Helps React optimize updates when answer lists change or reorder.
🔄 event handling	Managing user interactions like clicks, form submissions, and keyboard input in React components.	Your answer buttons use `onClick` handlers to trigger state changes and provide interactivity.

🤖 Ask the AI — Component Composition Mastery

You just built a complex interactive quiz system using component composition, array mapping, and dynamic styling — excellent work!

Now let's deepen your understanding of React patterns, component architecture, and user interaction design. Here are the most impactful questions to ask your AI assistant about today's session:

- **How does component composition make React apps more maintainable than monolithic components?**
- **How do conditional classes provide better user experience than static styling?**
- **How does the disabled attribute improve the user experience in quiz interfaces?**
- **Why is it better to store feedback messages in constants rather than hardcoding them?**
- **How does the AnswerChoices component demonstrate the single responsibility principle?**

💬 Pro Tip:

Component composition is the secret to building scalable React applications. Think of each component as having one clear job — AnswerChoices handles answer display, AnswerFeedback handles result messaging. This separation makes your code easier to test, debug, and extend.