Session 7 — Creating the Quiz Experience

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

☁️ Accessing Your Codespace

Visit [github.com/codespaces](#) to relaunch your Codespace from the previous session.

🧩 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)
```

Each component has a single responsibility, making your code easier to understand, test, and modify. Instead of one massive component doing everything, you break functionality into focused pieces that can be reused and debugged independently.

🔗 Connecting the Quiz Modal

🎯 **Goal:** Connect the quiz modal to your game flow so clicking zones displays quiz questions.

You'll set up the modal to appear when players click zones, creating the bridge between your game map and quiz questions.

Step 1: Add QuizModal to App

File: `src/App.jsx`

Make the QuizModal component available in your app so it can appear when players click zones.

```
// ... existing imports ...
import QuizModal from "./components/QuizModal";  // [1] Import QuizModal

export default function App() {
  const { screen, isQuizVisible } = useGame();   // [2] Add isQuizVisible

  return (
    <div className="app-container">
      {screen === SCREENS.SPLASH && <SplashScreen />}
      {screen === SCREENS.PLAYING && (
        <>
          <GameMap />
          <HUD />
          {isQuizVisible && <QuizModal />}
          {/* ↑ [3] Conditionally render QuizModal */}
          <CoordinateDisplay />
        </>
      )}
    </div>
  );
}
```

> 💡 **Understanding Conditional Rendering**
>
> 1. **Import QuizModal**: Import the QuizModal component at the top with other components
>
> 2. **Add isQuizVisible**: Add the modal visibility state to the destructuring
>
> 3. **Conditionally render QuizModal**: Use the `&&` operator to show QuizModal only when `isQuizVisible` is true
>
> The `&&` operator creates conditional rendering — when `isQuizVisible` is true, React renders `<QuizModal />`; when false, nothing renders. This pattern controls what users see based on app state.

Step 2: Update GameMap to show modal

File: `src/components/GameMap.jsx`

Make the modal appear after questions finish loading by updating the visibility state.

```
export default function GameMap() {
  // [1] Access setIsQuizVisible
  const { activeZone, loadQuestionsForZone, setIsQuizVisible, zoneProgress } =
    useGame();

  const handleZoneClick = async (zoneId) => {
    // ... existing code ...
    await loadQuestionsForZone(zoneId);
    setIsQuizVisible(true);  // [2] Show modal
  };

  // ... rest of component ...
}
```

> 💡 **Understanding Modal Visibility Control**
>
> 1. **Access setIsQuizVisible**: Get the setter function to control modal visibility
>
> 2. **Show modal**: Setting `isQuizVisible` to true satisfies Step 1's condition, rendering the modal
>
> This completes the connection: zone click → questions load → state updates to true → Step 1's condition satisfied → modal renders.

Step 3: Test the modal connection

Click any zone on the game map.

✓ **You should see:** QuizModal appears with your cached questions displayed!

🔵 Building Answer Buttons

🎯 **Goal:** Build interactive answer buttons that transform your question data into clickable choices using array mapping.

File: `src/components/QuizModal.jsx`

You'll create the `AnswerChoices` component in three steps: create the component structure, add it to JSX, then implement array mapping to generate buttons.

Step 1: Create AnswerChoices component

Create a skeleton version of the component that you'll progressively build up with functionality.

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

Step 2: Add AnswerChoices to QuizModal

Place the AnswerChoices component in your modal so it appears below the question.

```
export default function QuizModal() {
  // ... existing code ...

  return shouldShowModal ? (
    <div className="quiz-modal">
      <div className="quiz-content">
        {/* ... existing code ... */}

        <QuestionHeader question={question} />
        <AnswerChoices answers={question.answers} />
        {/* ↑ Add AnswerChoices component */}

        <ContinueButton
          hasAnswered={chosenAnswer !== null}
          isOnFinalQuestion={isLastQuestion}
          onContinue={handleContinue}
        />
      </div>
    </div>
  ) : null;
}
```

Step 3: Verify props are passed

Click a zone, then open React DevTools → Components tab → find AnswerChoices → check the `answers` prop.

✓ **You should see:** The `answers` prop is populated with an array of four answer strings.

Step 4: Generate buttons from answers array

Transform your answers array into clickable buttons using `map()` — one button for each answer.

```
function AnswerChoices({ answers }) {
  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (        // [1] Loop answers
        <button key={index} className="answer-button">  // [2] Create button
          {answer}                              // [3] Display answer
        </button>
      ))}
    </div>
  );
}
```

> 💡 **Understanding Array Mapping**
>
> 1. **Loop answers**: The `map()` method iterates through the answers array, converting each string into JSX
>
> 2. **Create button**: Each answer becomes its own button with a unique key for React's tracking
>
> 3. **Display answer**: The answer string appears as the visible text on each button
>
> 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.

Step 5: Test the answer buttons

Click a zone to open the quiz modal.

✓ **You should see:** Four answer buttons appear in the modal with your question's answer choices!

🎨 Making Buttons Interactive

🎯 **Goal:** Add click functionality and dynamic styling that shows correct/incorrect answers with visual feedback.

File: `src/components/QuizModal.jsx`

You'll add interactivity in three steps: click handling, conditional styling, and preventing multiple clicks.

Step 1: Add click handling to AnswerChoices

Part A: Update AnswerChoices component

Make your answer buttons respond to clicks by accepting a click handler prop.

File: `src/components/QuizModal.jsx`

```jsx
function AnswerChoices({ answers, onAnswerClick }) {  // [1] Add onAnswerClick prop
  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (
        <button
          key={index}
          className="answer-button"
          onClick={() => onAnswerClick(index)}  // [2] Attach click handler
        >
          {answer}
        </button>
      ))}
    </div>
  );
}
```

> 💡 **Understanding Click Handling**
>
> 1. **Add onAnswerClick prop**: Accept the click handler function from the parent component
> 2. **Attach click handler**: Add `onClick` to each button so it calls `onAnswerClick` with the button's `index`
>
> Each button needs to tell the parent component which answer was clicked. The `onClick` prop connects the button to the handler function, and the `index` parameter identifies which specific answer the player selected.

Part B: Pass the handler to AnswerChoices

Update the JSX to pass the `handleAnswerClick` function:

```
export default function QuizModal() {
  // ... existing code ...

  return shouldShowModal ? (
    <div className="quiz-modal">
      <div className="quiz-content">
        {/* ... existing code ... */}

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

        {/* ... rest of modal ... */}
      </div>
    </div>
  ) : null;
}
```

Part C: Test click handling

Click a zone, then click any answer button.

✓ **You should see:** The Continue button becomes enabled, allowing you to move to the next question.

Step 2: Add conditional styling to AnswerChoices

Part A: Update AnswerChoices component

Add visual feedback that shows which answer is correct and which was chosen incorrectly.

File: `src/components/QuizModal.jsx`

```jsx
// [1] Add new props
function AnswerChoices({ answers, onAnswerClick, chosenAnswer, correctAnswer }) {
  // [2] Create style function
  const getButtonStyle = (index) => {
    if (chosenAnswer === null) return "answer-button";
    if (index === correctAnswer) return "answer-button correct";
    if (index === chosenAnswer) return "answer-button incorrect";
    return "answer-button";
  };

  return (
    <div className="answers-grid">
      {answers.map((answer, index) => (
        <button
          key={index}
          className={getButtonStyle(index)}  // [3] Apply dynamic className
          onClick={() => onAnswerClick(index)}
        >
          {answer}
        </button>
      ))}
    </div>
  );
}
```

> 💡 **Understanding Dynamic Styling**
>
> 1. **Add new props**: Accept `chosenAnswer` and `correctAnswer` to track answer state
> 2. **Create style function**: `getButtonStyle()` returns different CSS classes based on state - default before answering, `correct` class for right answer, `incorrect` class for wrong answer, default for other buttons
> 3. **Apply dynamic className**: Each button calls the function with its `index` to get the appropriate styling
>
> The function checks conditions in order: if no answer is selected yet, all buttons are default. After selection, the correct answer gets distinct styling to show it was right, the chosen wrong answer gets different styling to show it was incorrect, and other buttons remain default. This provides immediate visual feedback.

Part B: Pass the styling props

Update the JSX to pass the styling props:

```jsx
export default function QuizModal() {
  // ... existing code ...

  return shouldShowModal ? (
    <div className="quiz-modal">
      <div className="quiz-content">
        {/* ... existing code ... */}

        <AnswerChoices
          answers={question.answers}
          onAnswerClick={handleAnswerClick}
          chosenAnswer={chosenAnswer}         {/* Add this prop */}
          correctAnswer={question.correct}    {/* Add this prop */}
        />

        {/* ... rest of modal ... */}
      </div>
    </div>
  ) : null;
}
```

Part C: Test conditional styling

Click a zone, then click an answer.

✓ **You should see:** The correct answer and your chosen answer (if incorrect) show distinct visual styling.

Step 3: Disable buttons after answer selection

Disable all buttons after an answer is selected so players can't change their answer.

```jsx
function AnswerChoices({ answers, onAnswerClick, chosenAnswer, correctAnswer }) {
  // ... existing code ...

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

Click a zone, click an answer, then try clicking other buttons.

✓ **You should see:** After selecting an answer, all other buttons become unclickable.

> 💡 **Preventing Multiple Clicks**
>
> The `disabled` attribute prevents multiple clicks after an answer is selected. Once `chosenAnswer` is no longer `null`, all buttons become disabled, creating a polished user experience where buttons respond intelligently to user interactions.

💬 Adding Feedback Messages

🎯 **Goal:** Add personality to your game with custom feedback messages that celebrate correct answers and encourage players after mistakes.

You'll create feedback message constants, import them, implement random selection logic, and add the feedback component to your quiz modal.

Step 1: Create the messages file

File: `src/constants/messages.js`

Create a new file to store your custom feedback messages for correct and incorrect answers.

Right-click `src/constants` → New File → name it `messages.js`, then add:

```
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!"
];
```

Step 2: Import the constants

File: `src/components/QuizModal.jsx`

Import your feedback messages so the QuizModal can use them.

```
// ... existing imports ...
import { CORRECT_FEEDBACK, INCORRECT_FEEDBACK } from "../constants/messages";
```

Step 3: Update AnswerFeedback component

File: `src/components/QuizModal.jsx`

Add logic that picks a random message from your feedback arrays each time a player answers.

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

  // [1] Pick the appropriate array
  const messages = isCorrect ? CORRECT_FEEDBACK : INCORRECT_FEEDBACK;
  // [2] Select random message
  const message = messages[Math.floor(Math.random() * messages.length)];

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

> 💡 **Understanding Random Selection**
>
> 1. **Pick the appropriate array**: Use ternary operator to choose CORRECT_FEEDBACK or INCORRECT_FEEDBACK
> 2. **Select random message**: `Math.random()` generates 0-1, multiply by array length, `Math.floor()` rounds down to get valid index
>
> This pattern adds variety to your game — players see different encouraging messages each time they answer, making the experience feel more dynamic and engaging.

Step 4: Add AnswerFeedback to QuizModal

File: `src/components/QuizModal.jsx`

Place the feedback component in your modal so players see their results after answering.

```
export default function QuizModal() {
  // ... existing code ...

  return shouldShowModal ? (
    <div className="quiz-modal">
      <div className="quiz-content">
        {/* ... existing code ... */}

        <QuestionHeader question={question} />

        <AnswerFeedback
          hasAnswered={chosenAnswer !== null}
          isCorrect={chosenAnswer === question.correct}
          correctAnswerText={question.answers[question.correct]}
        />
        {/* ↑ Add AnswerFeedback component */}

        {/* ... rest of modal ... */}
      </div>
    </div>
  ) : null;
}
```

Step 5: Test the feedback messages

Click a zone, then click different answers multiple times.

✓ **You should see:** Different feedback messages appear each time you answer, adding variety and personality to your game!

> 🏆 **Bonus Challenge**
>
> Add more feedback messages to each array to increase variety! Try different emojis and encouraging phrases.

🧪 Testing Your Quiz System

🎯 **Goal:** Verify your complete quiz system works by testing the full user flow and inspecting component state with DevTools.

You'll test the complete quiz flow from zone click to completion, then use React DevTools to inspect component state and props.

Step 1: Test complete quiz flow

Navigate to the game by clicking "Start Adventure", then test the full interaction sequence.

- **Click** any zone
- **Verify** modal appears with question and four answer buttons
- **Click** any answer
- **Verify**:
  - Button shows correct/incorrect styling
  - Random feedback message appears
  - Other buttons become disabled
  - Continue button becomes enabled
- **Click** Continue button
- **Verify** next question loads
- **Complete** all questions
- **Verify** modal closes and zone is marked complete in HUD

✓ **You should see:** The complete quiz flow works smoothly from start to finish, with all interactive elements responding correctly!

Step 2: Inspect component state with DevTools

Press `F12` or right-click → Inspect to open DevTools, then navigate to the Components tab.

- **Find** QuizModal in the component tree
- **Expand** to see child components
- **Click** AnswerChoices
- **Inspect** the `answers` prop (should show four answer strings)
- **Inspect** the `chosenAnswer` prop (should be `null` before answering)
- **Click** an answer in the game
- **Observe** `chosenAnswer` updates to the selected index

✓ **You should see:** React DevTools shows how props and state flow through your component tree, updating in real-time as you interact with the quiz!

> 💡 **End-to-End Testing**
>
> 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. React DevTools gives you X-ray vision into your component hierarchy, letting you verify that data flows correctly from parent to child 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 — Creating the Quiz Experience

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?