Session 9 — Adding Theme Music

You're about to add another dimension to your trivia game — theme music! This guide walks you through creating custom React hooks, working with browser audio APIs, and building reusable audio controls. Ready to bring your game to life with sound? Let's go!

Table of Contents

☁️ Accessing Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 8.

🪝 Custom Hooks

Before we dive into audio, let's understand **custom hooks** — one of React's most powerful patterns for code reuse.

Custom hooks are functions that start with "use" and let you extract component logic into reusable functions. Think of them as your own personal React features that you can use across multiple components.

Built-in Hooks	Custom Hooks
`useState`, `useEffect`, `useRef`	`useGame`, `useAudio`
Provided by React	Created by developers
Basic React features	Complex, reusable logic
Used in every React app	Specific to your app's needs

Custom hooks embody one of programming's most important principles: **"Don't Repeat Yourself" (DRY)**. Instead of copying and pasting audio logic into every component that needs music, you encapsulate that complexity into a reusable `useAudio` hook. Now, any component can add background music with a single line of code:

```
const music = useAudio('/music.mp3');
```

This is the difference between basic and advanced code — experienced developers reduce repetition by creating reusable solutions. Custom hooks let you "write once, use often," making your code cleaner, more maintainable, and easier to scale.

> 🏆 **Bonus Challenge**
>
> Visit useHooks – The React Hooks Library and find three interesting custom hooks that might be great for use in a web-based video game.

🔊 Browser Audio APIs

Let's understand the **HTMLAudioElement** — the browser's built-in interface for controlling audio playback.

You might be familiar with HTML audio elements like `<audio src="music.mp3"></audio>` that you write in HTML files. HTMLAudioElement is the JavaScript version of the same thing — it's like a digital music player that you create and control entirely with JavaScript code. Instead of writing HTML tags, you use `new Audio()` to create the player, then control it with methods like `play()` and `pause()`.

Creating Audio Elements

```
// Create new audio element
const audio = new Audio(getAssetPath('audio/theme-music.mp3'));

// Configure audio properties
audio.loop = true;        // Repeat when finished
audio.volume = 0.5;       // 50% volume
```

Audio Control Methods

```
// Playback control
audio.play();             // Start playing
audio.pause();            // Stop playing

// Properties
audio.currentTime = 0;    // Reset to beginning
audio.muted = true;       // Mute audio
```

The HTMLAudioElement gives you programmatic control over audio playback. Your `useAudio` hook will wrap this browser API in a clean React interface, making it easy to add music to any component.

🔗 Refs and useRef

Now let's understand **refs** — React's way to "step outside" the component system and work directly with DOM elements or browser APIs.

Refs are like bookmarks that let you remember information that doesn't affect what's rendered on the page. Unlike state, changing a ref doesn't trigger re-renders.

State	Refs
Triggers re-renders when changed	No re-renders when changed
For data that affects UI	For data that doesn't affect UI
`const [value, setValue] = useState()`	`const ref = useRef()`

Access with `value`	Access with `ref.current`

Common useRef Patterns

Storing Mutable Values (Your Audio Use Case):

```
function useAudio(src) {
  const audioRef = useRef(null); // Starts as null, won't trigger re-renders

  const play = () => {
    if (!audioRef.current) { // Check if audio element exists
      audioRef.current = new Audio(src); // Store in .current property
    }
    audioRef.current.play(); // Access stored element via .current
  };
}
```

The ref acts like a bookmark — it remembers where your audio element is so you can find it again later. Create the audio element once, bookmark it in `audioRef.current`, then use that bookmark every time `play()` is called. No re-renders, no recreating the same audio element.

Accessing DOM Elements:

```
function MyComponent() {
  const inputRef = useRef(null); // Create ref for DOM element

  const focusInput = () => {
    inputRef.current.focus(); // Call DOM method via .current
  };

  return <input ref={inputRef} />; // Connect ref to DOM element
}
```

The ref creates a direct connection to the actual HTML input element. When you call `inputRef.current.focus()`, you're telling the browser to focus that specific input — just like clicking on it.

Refs are perfect for storing audio elements because the audio object doesn't need to trigger re-renders — it just needs to be remembered between function calls. The `current` property holds the actual value you stored.

🎛️ Building the MusicToggle Component

🎯 **Goal:** Add the UI controls you'll need to test the audio functionality as you build it.

This music toggle will provide the interface for testing the `useAudio` hook as you implement it in the next sections.

Step 1: Add asset utility import

File: `src/components/HUD.jsx`

Add the asset utility import at the top of the file.

```
import { getAssetPath } from "../utils/assets";
```

Step 2: Create MusicToggle component

File: `src/components/HUD.jsx`

Add the MusicToggle component after the CurrentZone function.

```
function MusicToggle() {
  const { music } = useGame();
  return (
    <button
      onClick={music.toggle}
      className="music-toggle"
      title={music.isPlaying ? "Pause Music" : "Play Music"}
    >
      <img
        src={getAssetPath(
          music.isPlaying ? "images/playing.svg" : "images/paused.svg"
        )}
        alt={music.isPlaying ? "Pause" : "Play"}
        className="music-icon"
        width={24}
        height={24}
      />
    </button>
  );
}
```

Step 3: Add MusicToggle to HUD

File: `src/components/HUD.jsx`

Update the HUD JSX return to include the MusicToggle component.

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

Step 4: Test music toggle visibility

Start the game and navigate to the playing screen.

✓ **You should see:** Music toggle button is visible but inoperable when clicked (audio functionality not yet implemented).

> 💡 Conditional Rendering with Dynamic Content
>
> The MusicToggle component demonstrates conditional rendering with dynamic images and tooltips. The `music.isPlaying` state controls both the icon and the tooltip text, providing clear visual feedback to users. This pattern — using state to drive multiple UI elements — creates cohesive, responsive interfaces.

📻 Adding Audio Reference to useAudio

🎯 **Goal:** Add the audio reference to your `useAudio` hook so it can store the HTMLAudioElement.

You'll add a ref to store the audio element and import the necessary React hooks.

Step 1: Add useRef import

File: `src/hooks/useAudio.js`

Add the useRef import at the top of the file.

```
import { useRef, useState } from "react";
```

Step 2: Add audio reference to hook

File: `src/hooks/useAudio.js`

Add the audio reference inside the `useAudio` function.

```
export function useAudio(src) {
  const audioRef = useRef(null);
  const [isPlaying, setIsPlaying] = useState(false);
  // ... rest of hook
}
```

Your hook now has a ref that can store and remember the audio element you'll create in the `play` function. The ref starts as `null` and will hold your audio element once it's created.

Audio Reference Flow

```
useAudio hook called → audioRef.current is null → play() creates new Audio() →
audioRef.current stores Audio element → future calls reuse same element
```

> 💡 Persistent Storage Across Re-renders
>
> The `audioRef` you just created provides persistent storage for the audio element across component re-renders. Without refs, you'd create a new audio element every time the component updates, causing audio to restart unexpectedly. Refs solve this by maintaining the same reference to the audio object throughout the component's lifecycle.

🎵 Implementing Audio Playback

🎯 **Goal:** Implement the core audio functionality by updating the `play` function to create and control audio elements.

You'll add lazy initialization to create the audio element only when needed, then configure and play it.

Step 1: Update play function

File: `src/hooks/useAudio.js`

Update the play function to create and control the audio element.

```
const play = () => {
  // [1] Lazy initialization
  if (!audioRef.current) {
    audioRef.current = new Audio(src);
    audioRef.current.loop = true;
    audioRef.current.volume = 0.5;
  }
  // [2] Play audio
  audioRef.current.play();
  // [3] Update state
  setIsPlaying(true);
};
```

> 💡 Understanding Lazy Initialization
>
> 1. **Lazy initialization**: Create audio element only when first needed — the `if (!audioRef.current)` check ensures the audio element is created once and reused
> 2. **Play audio**: Call the browser's play method to start playback
> 3. **Update state**: Set `isPlaying` to true so UI reflects the playing state
>
> The `if (!audioRef.current)` check is an example of **lazy initialization** — creating a resource only when it's first needed. Since `audioRef.current` starts as `null`, the first time `play()` runs it creates the audio element. Every time after that, `audioRef.current` contains the audio element, so the `if` condition is false and it skips creating a new one.

Step 2: Test audio playback

Click the music toggle button.

✓ **You should see:** Game theme plays and button shows playing state (icon changes).

> 💡 Preventing Audio Chaos
>
> Creating audio elements only once and reusing them prevents overlapping sounds, memory leaks, and performance issues. Without this pattern, clicking the music toggle rapidly would create multiple audio elements playing simultaneously, causing audio chaos and slowing down your browser. The lazy initialization pattern ensures clean, efficient audio management.

⚡ GitHub Copilot Workflow

You're now working with production-quality code. GitHub Copilot can help you write, fix, and understand code faster — but only if you know how to guide it.

How to Use Copilot Chat Effectively

1. **Use** a Copilot chat command like `/fix`, `/explain`, or `/test`
2. **Write** a clear, focused prompt describing what you want
3. **Review** the suggestion Copilot generates
4. **Apply** the change if it meets your needs
5. **Test** the update to confirm it works

Example Prompt

> 🤖 AI Prompt:
>
> ```
> /fix Add error handling to the play function in the useAudio hook so that if the au
> ```

Use this workflow during your Solo Mission and anytime you're stuck or want to improve your code.

🎖️ Solo Mission: Complete useAudio Hook

Now for your independent challenge — complete the `useAudio` hook with pause functionality, error handling, and cleanup using AI assistance.

What You're Building

A complete custom audio hook that manages background music playback with play/pause controls, error handling, and proper cleanup. This hook will encapsulate all audio logic, making it reusable across any component that needs music.

Phase 1: Pause Functionality

🎯 **Goal:** Add pause controls to stop audio playback and update state

Your Task:

In `src/hooks/useAudio.js`, update the `pause` function to:

- Call `audioRef.current.pause()` method
- Set `isPlaying` to `false`

Test: Click the music toggle while audio is playing → music should stop and the icon should switch to the paused state

---

Phase 2: Error Handling

🎯 **Goal:** Prevent audio failures from crashing the app

Your Tasks:

1. With `src/hooks/useAudio.js` open, issue the following prompt to GitHub Copilot:

   > 🤖 **AI Prompt:**
   >
   > `/fix Add error handling to the play function in the useAudio hook so that if the`

2. Review the generated code

3. Apply the changes if they look correct

Test error handling:

1. Open `src/context/GameContext.jsx`

2. Locate

```
const music = useAudio(getAssetPath("audio/dramatic-action.mp3"));
```

3. Change the path to `"audio/nonexistent.mp3"` (keep the `getAssetPath()` wrapper)
4. Click music toggle → check browser console for an error message
5. Revert the path back to `"audio/dramatic-action.mp3"` after testing

✓ **You should see:** Console shows error message, app doesn't crash, and `isPlaying` stays false

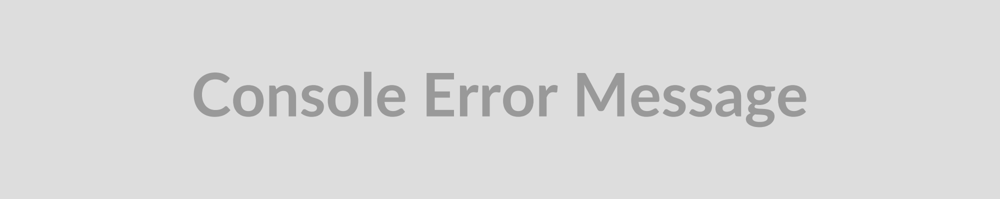

Figure: Error handling prevents crashes and logs helpful debugging information

---

Phase 3: Cleanup Function

🎯 **Goal:** Prevent memory leaks by cleaning up when the hook unmounts

Your Tasks:

1. With `src/hooks/useAudio.js` open, issue the following prompt to GitHub Copilot:

> 🤖 **AI Prompt:**
>
> `/fix Add a useEffect cleanup function to the useAudio hook that stops the audio`

2. Review the generated `useEffect` code
3. Apply the changes

Verify your implementation:

Check that your `useAudio` hook includes a `useEffect` with a cleanup function that:

- Pauses the audio if it's playing
- Clears the `audioRef.current` reference
- Returns the cleanup function from `useEffect`

✅ Success Review

Your completed `useAudio` hook should:

- Export `play`, `pause`, `toggle`, and `isPlaying`
- Successfully toggle audio playback on/off
- Handle errors when attempting to play audio
- Set `isPlaying` to false if an error occurs
- Include a `useEffect` cleanup function for component unmounting
- Prevent memory leaks when navigating between screens

🔍 Reference Guide

- **src/hooks/useGame.js** – Custom hook structure, exporting functions and state
- **src/components/QuizModal.jsx** – useEffect cleanup pattern
- **src/context/GameContext.jsx** – Error handling with try/catch blocks

📚 Essential Terms

Quick reference for all the custom hooks and browser API concepts you just learned:

Term	Definition	Why it matters
🪝 custom hook	Functions starting with "use" that let you extract and reuse component logic across multiple components.	Hooks like `useAudio` let you "write once, use often" instead of copying audio logic across components.
🔄 DRY (Don't Repeat Yourself)	A fundamental programming principle that emphasizes eliminating code duplication through reusable solutions.	Custom hooks embody DRY by encapsulating complex logic into reusable functions.

🔊 HTMLAudioElement	Part of the Web API that provides an interface for controlling audio playback, with methods like play(), pause(), and properties like volume and loop.	Gives you programmatic control over audio files in web applications.
🔗 ref	A way to access DOM elements or store values that don’t cause re-renders when changed.	Perfect for storing audio elements that need to persist but don’t affect UI rendering.
🎯 useRef	A React hook that creates a persistent reference to a DOM element or value that doesn’t cause re-renders when it changes.	Essential for storing audio elements and other browser API objects across component updates.
🔄 mutable	Data that can be changed or modified after it’s created, as opposed to immutable data that cannot be changed.	Refs store mutable values that can be updated without triggering re-renders, perfect for audio objects that change state.
⏳ lazy initialization	A pattern where resources are created only when first needed, rather than upfront.	Avoids unnecessary setup and ensures efficient resource reuse, like creating audio elements only when play() is first called.

🤖 Ask the AI — Adding Theme Music

You just created a custom React hook with browser API integration and AI-assisted development — excellent work!

Now let's deepen your understanding of custom hooks, browser APIs, and development practices. Here are the most impactful questions to ask your AI assistant about today's session:

- **What makes custom hooks different from regular functions, and why do they need to start with "use"?**
- **How do refs differ from state, and when should I use each one?**
- **Why do I need to use ref.current instead of just ref?**
- **What are the benefits of wrapping browser APIs in custom hooks?**
- **How does HTMLAudioElement work, and what other Web APIs are commonly used in web development?**
- **How can AI assistants help with coding, and what should I watch out for?**