Session 9 — Adding Theme Music

Custom Hooks & Browser APIs 🎵

You're about to add another professional feature 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

☁️ Access Your Codespace

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

🪝 Understanding 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 vs Custom Hooks

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

💡 Why This Matters

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 amateur and professional code — professionals 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.

📢 Exploring 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
```

💡 Why This Matters

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.

🔗 Understanding 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 vs Refs: The Key Differences

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.

💡 Why This Matters

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

Before we implement the audio functionality, let's add the UI controls you'll need to test it. This music toggle will provide the interface for testing the `useAudio` hook as you build it in the next sections.

1. **Add the asset utility import** at the top of `src/components/HUD.jsx`:

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

2. **Add the `MusicToggle` component** after the `CurrentZone` function:

```
function MusicToggle() { // Add MusicToggle component
  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>
  );
}
```

3. **Add `MusicToggle` to the HUD** by updating the JSX return:

```
return (
  <>
    <Scoreboard />
    <CurrentZone />
    <MusicToggle /> // Add MusicToggle
  </>
);
```

4. **Test**: Start Game → Music toggle visible, but inoperable when clicked

💡 Why This Matters

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.

📻 Adding Audio Reference to useAudio

Now let's add the audio reference to your `useAudio` hook so it can store the HTMLAudioElement.

1. **Open** `src/hooks/useAudio.js` and add the audio reference:

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

2. **Add the useRef import** at the top of the file:

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

Your hook now has a ref that can store and remember the audio element we'll create in the `play` function. The ref starts as `null` and will hold our 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
```

💡 Why This Matters

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.

🎵 Implementing Audio Playback

Let's implement the core audio functionality by updating the `play` function to create and control audio elements.

1. **Update the play function** in `src/hooks/useAudio.js`:

```
const play = () => { // Update play function
  if (!audioRef.current) {
    audioRef.current = new Audio(src);
    audioRef.current.loop = true;
    audioRef.current.volume = 0.5;
  }
  audioRef.current.play();
  setIsPlaying(true);
};
```

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.

2. **Test**: Click music toggle → Game theme plays and button shows playing state

Audio Creation Logic

```
Check if audio exists → If not, create new Audio(src) → Configure loop and volume → Call play() method → Update isPlaying state → UI reflects playing state
```

💡 Why This Matters

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.

⚡ 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

```
/fix Add error handling to the play function in the useAudio hook so that
if the audio fails to play, it catches the error, logs a warning, and
updates isPlaying to false
```

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! You'll use AI assistance for the advanced parts.

1. Implement Pause Functionality

- Update the `pause` function to pause audio and update state
- Use `audioRef.current.pause()` to stop playback
- Set `isPlaying` to `false` when paused
- **Test**: Click music toggle while playing → Music stops and music toggle shows paused state

2. Add Error Handling with AI Assistance

With `useAudio.js` open, use the GitHub Copilot workflow you just learned:

Prompt:

```
/fix Add error handling to the play function in the useAudio hook so that
if the audio fails to play, it catches the error, logs a warning, and
updates isPlaying to false
```

- Review the generated code
- Apply the changes if they look correct
- **Test**: To verify error handling works, temporarily break the audio path:

- Open `src/context/GameContext.jsx`
- Find the line:
  `const music = useAudio(getAssetPath("audio/dramatic-action.mp3"));`
- Change `"audio/dramatic-action.mp3"` to `"audio/nonexistent.mp3"` (keep the `getAssetPath()` wrapper)
- Save the file → Click music toggle → Check browser console for error message
- **Important**: Change the path back to `"audio/dramatic-action.mp3"` when done testing

3. Add Cleanup with AI Assistance

Prompt:

```
/fix Add a useEffect cleanup function to the useAudio hook that stops
the audio and clears the reference when the component unmounts
```

- Review the generated useEffect code
- Apply the changes to prevent memory leaks
- **Test**: Navigate between screens to verify cleanup works

Requirements Checklist

Your completed `useAudio` hook must:

- Export `play`, `pause`, `toggle`, and `isPlaying`
- Successfully toggle audio playback
- Handle errors when attempting to play audio
- Set `isPlaying(false)` if an error occurs
- Include a `useEffect` cleanup function for component unmounting

💡 Why This Matters

This challenge combines everything you've learned: custom hooks, browser APIs, error handling, and AI-assisted development. You're building production-quality code that handles edge cases and prevents memory leaks — exactly what professional developers do.

📚 Essential Terms

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

Term	Definition	Why it matters
🪝 hook	Functions starting with “use” that let you use React features like state and context.	Hooks like `useState` are your tools for managing data and behavior in components.
🔄 DRY (Don’t Repeat Yourself)	A fundamental programming principle that emphasizes eliminating code duplication through reusable solutions.	Custom hooks like `useAudio` let you “write once, use often” instead of copying audio logic across components.
🔊 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 — Custom Hooks & Browser APIs

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 professional 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?**