Session 4 — Configuring Game Zones

You're about to design the heart of your trivia game — the zone configuration that defines your entire game experience! This guide walks you through creating cohesive zone themes, understanding JavaScript data structures, and building the metadata that powers your adventure. Ready to architect your game world? Let's go!

Table of Contents

☁️ Accessing Your Codespace

Visit github.com/codespaces to relaunch your Codespace from Session 3.

🖥️ Adding the HUD and Coordinate Display

🎯 **Goal:** Add the game's HUD (Heads Up Display) and coordinate helper to prepare for zone configuration.

File: `src/App.jsx`

Step 1: Add imports to App

Import the HUD and coordinate display components you'll add to the game screen.

```
// Add these two imports
import CoordinateDisplay from "./components/CoordinateDisplay";
import HUD from "./components/HUD";
```

Step 2: Update PLAYING screen rendering

Add the HUD and coordinate display alongside your game map.

```
// Before:
{screen === SCREENS.PLAYING && <GameMap />}

// After:
{screen === SCREENS.PLAYING && (
  <>
    <GameMap />
    <HUD />
    <CoordinateDisplay />
  </>
)}
```

Step 3: Test the display

Navigate to the game screen by clicking “Start Adventure.”

✓ **You should see:** The game map with a HUD at the top showing zone progress and a coordinate display in the corner showing your mouse position.

> 💡 Fragments Group Without Clutter
>
> **React Fragments** (`<>...</>`) are like invisible containers that let you group multiple components without adding extra DOM elements. React requires a single root element, so fragments solve this cleanly. The HUD shows game progress, while CoordinateDisplay helps you position zone labels precisely in the next sections.

> 🏆 Bonus Challenge
>
> Try removing the fragment tags (`<>` and `</>`) and see what error React gives you when trying to return multiple elements!

📋 Metadata and Configuration

Before diving into code, let's understand what makes your game tick — **metadata** and **configuration files** that define your entire game experience.

What is Metadata?

Metadata is data about data. Think of it like a restaurant menu: it tells you everything about the dish — name, price, ingredients, spice level — but it's not the actual food. Your zone metadata works the same way: it describes each zone's properties without being the actual trivia questions.

Configuration Files and Data-Driven Architecture

Configuration files like `zones.js` separate data from code. Your `zones.js` file contains all the information that defines how your zones look, behave, and connect to trivia content. This makes your app flexible and easy to modify without touching component code.

Example: Want to change a zone's difficulty from "easy" to "medium"? Just update one value in `zones.js`. Want to add a fourth zone? Add another object to the array. The components automatically adapt to your data.

> 💡 Data-Driven Design
>
> Separating data from code is a fundamental pattern in software development. When your app reads configuration data, you can modify the entire game experience by editing a single file. This same pattern powers everything from video games to web applications — the code stays the same, but the data defines what users experience.

🏗️ The ZONES Array Structure

Let's examine the **data structures** that power your zone configuration — **arrays** and **objects**.

File: `src/data/zones.js`

Here's the ZONES array structure that defines your game:

```
export const ZONES = [
  {
    id: 0,                                   // ← Zone number for tracking
    name: "Binary Woods",                    // ← Zone title shown to players
    subtitle: "Bytes & Bugs",                // ← Zone description
    categoryId: 18,                          // ← Trivia category (Computers)
    difficulty: "easy",                      // ← Question difficulty level
    questionCount: 4,                        // ← How many questions in this zone
    mapLabel: {                              // ← Nested object for positioning
      x: 225,                                // ← Horizontal position on map
      y: 140,                                // ← Vertical position on map
      fontSize: "35",                        // ← Label text size
      fontFamily: "Pirata One, serif",       // ← Label font style
      color: "#333",                         // ← Label text color
      fontWeight: "normal",                  // ← Label font weight
      alignment: "left",                     // ← Label text alignment
    },
  },
  // Zone 1 and Zone 2 follow the same structure...
];
```

Understanding the Structure

Arrays (the `[]` brackets) are ordered lists — perfect for your three zones that players complete in sequence. Arrays use zero-based indexing, so Zone 0 is first, Zone 1 is second, and Zone 2 is third.

Objects (the `{}` curly braces) are collections of key-value pairs — ideal for storing all the details about one zone. Each property uses colon syntax (`name: "value"`) and properties are separated by commas.

The `ZONES` array will contain three zone objects — one for each game zone — each containing a **nested** `mapLabel` object inside it. This combination of arrays and objects lets you represent complex, real-world data in code.

Data Types in Your Configuration

- **Strings** (in quotes): `"Binary Woods"`, `"easy"` — text data
- **Numbers** (no quotes): `18`, `4`, `225` — numeric data for calculations and positioning
- **Objects** (inside objects): `mapLabel: { x: 225, y: 140 }` — complex data with multiple properties

> 💡 Arrays and Objects Work Together
>
> Arrays are like playlists — they keep things in order. Objects are like contact cards — they store all the details about one thing. Together, they're the perfect combination for organizing your game world. You'll see this pattern everywhere in JavaScript: arrays of objects representing lists of complex items.

🎨 Designing Your Zone Themes

Now for the creative part — designing three distinctive zone themes that create a cohesive game experience!

Step 1: Explore available categories

Visit https://opentdb.com/api_category.php to see all available trivia categories and their IDs.

Step 2: Brainstorm zone concepts

Think about themes that match the visual environments on your game map:

- **Forest theme**: Nature, animals, science categories
- **Desert theme**: History, geography, mythology categories
- **Ice castle theme**: Entertainment, sports, art categories

Step 3: Plan your zones

Use this organizer to capture your design decisions for each zone:

Zone 0 (First zone):

- **name**: ______________________________
- **subtitle**: ______________________________
- **categoryId**: _____ (from API categories)
- **difficulty**: _____ (easy, medium, or hard)
- **questionCount**: _____ (max 50)

Zone 1 (Second zone):

- name: ___________________________
- subtitle: ___________________________
- **categoryId**: _____ (from API categories)
- **difficulty**: _____ (easy, medium, or hard)
- **questionCount**: _____ (max 50)

Zone 2 (Third zone):

- name: ___________________________
- subtitle: ___________________________
- **categoryId**: _____ (from API categories)
- **difficulty**: _____ (easy, medium, or hard)
- **questionCount**: _____ (max 50)

Design Tips

- **Create progression**: Start with easier categories and increase difficulty
- **Match themes**: Choose categories that fit the visual environment
- **Balance length**: Vary question counts to create pacing (e.g., 4, 7, 10 questions)
- **Consider your audience**: Pick categories that interest you and your players

> 💡 Cohesive Theming Creates Immersion
>
> Matching visual environments with appropriate trivia categories creates logical connections that help players navigate and remember your game. When the "Forest of Knowledge" asks science questions and the "Desert of History" asks about ancient civilizations, players intuitively understand your game world. This attention to user experience separates good games from great ones.

⚙️ Configuring Zone Metadata

🎯 **Goal:** Implement your zone designs by updating the ZONES array with your custom metadata from your planning organizer.

File: `src/data/zones.js`

Step 1: Update Zone 0

Replace the first zone object with your Zone 0 design from your planning organizer. Use the values you decided on for `name`, `subtitle`, `categoryId`, `difficulty`, and `questionCount`. Keep the `mapLabel` coordinates as-is for now.

Example structure (use YOUR values from the organizer):

```
{
  id: 0,
  name: "Your Zone Name",          // ← Use your Zone 0 name
  subtitle: "Your Zone Subtitle",    // ← Use your Zone 0 subtitle
  categoryId: 18,                  // ← Use your Zone 0 categoryId
  difficulty: "easy",              // ← Use your Zone 0 difficulty
  questionCount: 4,                // ← Use your Zone 0 questionCount
  mapLabel: {
    x: 225,
    y: 140,
    fontSize: "35",
    fontFamily: "Pirata One, serif",
    color: "#333",
    fontWeight: "normal",
    alignment: "left",
  },
},
```

Step 2: Add Zone 1

Add a second zone object after Zone 0 with your Zone 1 design from your planning organizer. You can copy Zone 0's structure and modify the values, or type it from scratch.

Example structure (use YOUR values from the organizer):

```
{
  id: 1,
  name: "Your Zone Name",          // ← Use your Zone 1 name
  subtitle: "Your Zone Subtitle",    // ← Use your Zone 1 subtitle
  categoryId: 9,                   // ← Use your Zone 1 categoryId
  difficulty: "medium",            // ← Use your Zone 1 difficulty
  questionCount: 7,                // ← Use your Zone 1 questionCount
  mapLabel: {
    x: 360,
    y: 530,
    fontSize: "35",
    fontFamily: "Pirata One, serif",
    color: "#333",
    fontWeight: "normal",
    alignment: "left",
  },
},
```

Step 3: Add Zone 2

Add a third zone object after Zone 1 with your Zone 2 design from your planning organizer. Again, you can copy an existing zone's structure and modify the values.

Example structure (use YOUR values from the organizer):

```
{
  id: 2,
  name: "Your Zone Name",          // ← Use your Zone 2 name
  subtitle: "Your Zone Subtitle",    // ← Use your Zone 2 subtitle
  categoryId: 17,                  // ← Use your Zone 2 categoryId
  difficulty: "hard",              // ← Use your Zone 2 difficulty
  questionCount: 10,               // ← Use your Zone 2 questionCount
  mapLabel: {
    x: 1000,
    y: 400,
    fontSize: "35",
    fontFamily: "Pirata One, serif",
    color: "#333",
    fontWeight: "normal",
    alignment: "middle",
  },
},
```

Step 4: Test your configuration

Navigate to the game screen by clicking "Start Adventure."

✓ **You should see:** Your custom zone names and subtitles appear on the game map (though positions may need adjustment in the next section).

Note: Changes to `zones.js` trigger a full page reload, not hot module replacement.

> 💡 Configuration as Game Design
>
> By editing this single data file, you're designing your entire game experience without touching any component code. This separation means you could create completely different games — a history quiz, a science challenge, a pop culture trivia — just by changing the data in `zones.js`. The components stay the same; the data defines what players experience. This is the power of data-driven architecture in action.

📍 Positioning Zone Labels

🎯 **Goal:** Use the CoordinateDisplay component to find optimal positions for your zone labels on the game map.

File: `src/data/zones.js`

Step 1: Navigate to the game screen

Click "Start Adventure" to see your game map with the coordinate display in the corner.

Step 2: Find optimal positions

Move your mouse around the map and watch the coordinate display show real-time x and y values. Find good positions for each zone label that don't overlap with visual elements or other labels.

Tips for positioning:

- Hover over different areas of each zone
- Note coordinates where labels would be clearly visible
- Avoid edges and busy visual areas
- Consider label alignment (left, middle, right)

Step 3: Update Zone 0 coordinates

Update the `x` and `y` values in Zone 0's `mapLabel` object with your chosen coordinates.

```
mapLabel: {
  x: 225,  // ← Update with your x coordinate
  y: 140,  // ← Update with your y coordinate
  // ... other styling properties
}
```

Step 4: Update Zone 1 coordinates

Update the `x` and `y` values in Zone 1's `mapLabel` object.

```
mapLabel: {
  x: 360,  // ← Update with your x coordinate
  y: 530,  // ← Update with your y coordinate
  // ... other styling properties
}
```

Step 5: Update Zone 2 coordinates

Update the `x` and `y` values in Zone 2's `mapLabel` object.

```
mapLabel: {
  x: 1000,  // ← Update with your x coordinate
  y: 400,   // ← Update with your y coordinate
  // ... other styling properties
}
```

Step 6: Test and refine

Let the page reload and check your label positions. Adjust coordinates as needed until all labels are clearly visible and well-positioned.

✓ **You should see:** All three zone labels positioned exactly where you want them on the map.

> 💡 Tool-Assisted Design
>
> The CoordinateDisplay component demonstrates a key principle in development: build tools that make your work easier. Instead of guessing coordinates and refreshing repeatedly, you get instant feedback. This "observe, note, update, verify" workflow appears everywhere in game development and UI design — from level editors to animation tools. Good developers build tools to solve problems once, then reuse them.

> 🏆 Bonus Challenge
>
> Try experimenting with other mapLabel properties like `fontSize`, `color`, or `alignment` to customize your zone labels' appearance!

🔍 Testing with React DevTools

🎯 **Goal:** Use React DevTools to explore your zone configuration and test game progression without playing through the entire game.

Step 1: Open DevTools

Press `F12` or right-click → Inspect to open your browser's developer tools.

Step 2: Navigate to Components tab

Look for "Components" next to Console, Network, etc. and click it.

✓ **You should see:** A tree of React components.

Step 3: Locate GameProvider

Click on `GameProvider` in the component tree.

✓ **You should see:** The right panel shows GameProvider's hooks and state.

Step 4: Find zoneProgress state

Look for the `zoneProgress` state in the hooks section. If hook names aren't clear, click the gear icon and enable "Parse hook names."

✓ **You should see:** An array with three objects, one for each zone, showing `completed` and `score` properties.

Step 5: Experiment with progression

Change the first zone's `completed` property from `false` to `true` and watch the UI update.

✓ **You should see:** The HUD updates to show Zone 0 as completed, and `activeZone` and `currentZone` values change automatically.

Step 6: Observe state relationships

Notice how changing one piece of state (`zoneProgress`) automatically updates related values (`activeZone`, `currentZone`) and the UI reflects these changes instantly.

✓ **You should see:** The entire app responds to the state change — this is the power of shared state management!

> 💡 X-Ray Vision for Your App
>
> React DevTools gives you superpowers: inspect any component's data, then manipulate it instantly to test complex scenarios. Want to see what happens when all zones are complete? Change the state. Curious about edge cases? Modify values and watch. This transforms debugging from guesswork into precision — test interactions, simulate states, and verify data flow without writing test code.

📚 Essential Terms

Quick reference for all the data structure and configuration concepts you just learned:

Term	Definition	Why it matters
📋 metadata	Data that describes other data — information about information.	Your zone configuration describes how to get and display trivia questions without being the questions themselves.

🏗️ array	An ordered list of items using bracket syntax `[]` with zero-based indexing.	Perfect for storing your three game zones in a specific order that matches the game progression.
📦 object	A collection of key-value pairs using curly brace syntax `{}` with colon-separated properties.	Ideal for zone properties like `name`, `difficulty`, and styling — each zone is an object with multiple attributes.
🔤 string	Text data enclosed in quotes, used for names, descriptions, and categories.	Zone names, subtitles, and difficulty levels are all strings that display to users.
🔢 number	Numeric data without quotes, used for IDs, counts, and coordinates.	Category IDs, question counts, and map coordinates are numbers used for calculations and positioning.
🏷️ property	A key-value pair within an object, accessed using dot notation like `zone.name`.	Each zone object has properties like `name`, `categoryId`, and `mapLabel` that define its characteristics.
⚛️ React Fragment	JSX syntax `<>...</>` that groups elements without adding extra DOM nodes.	Lets you return multiple components from the PLAYING screen without wrapper divs cluttering your HTML.

🤖 Ask the AI — Configuring Game Zones

You just designed cohesive zone themes, configured complex JavaScript data structures, and experienced data-driven architecture — excellent work!

Now let's deepen your understanding of data structures, configuration patterns, and the relationship between data and UI. Here are the most impactful questions to ask your AI assistant about today's session:

- What makes arrays and objects different, and when should I use each?
- How does nesting data structures help represent complex real-world information?
- Why is separating data from code a good practice in software development?
- What are the benefits of using configuration files like zones.js?
- How do React Fragments solve the single root element requirement?
- What is metadata and why is it important in application design?
- How does changing data in zones.js affect what users see without modifying components?