**Casino Content Portal: Full Stack Developer Test**

Estimated Time: 3 hours (+/- 30 minutes)

Objective: To build a simple full-stack application that displays casino content (games, promotions, news) from a static data source, implements client-side filtering/search, and demonstrates core React, Express, and TypeScript skills with basic unit testing.

Introduction:

Welcome to your Full Stack Developer Test. This task is designed to be completed within a focused 3-hour window. We understand that this is a tight timeframe, so we prioritize core functionality, clean code, thoughtful architecture, and effective problem-solving over exhaustive features or pixel-perfect design. Your ability to prioritize and deliver a functional solution within the given constraints is what we're assessing.

You will create a simple backend API to serve content and a React frontend to display and interact with this content.  
  
**Important Note on AI Tool Usage:**  
We encourage the use of development tools, including AI-powered assistants (e.g., GitHub Copilot, ChatGPT) for support, clarification, or brainstorming. However, **the code you submit must be your own work, and you must thoroughly understand every line of code in your implementation.** Directly generated, large blocks of code from AI tools without your full comprehension or significant modification are not permitted. The purpose of this test is to evaluate *your* coding skills and understanding.

Key Technologies:

* Frontend: ReactJS, TypeScript, HTML, CSS (any styling approach is fine - e.g., plain CSS, SASS, CSS Modules, styled-components).
* Backend: Express, TypeScript.
* Testing: Unit Testing (e.g., Jest or React Testing Library).

I. Provided Data

You have been provided with the casinoContent.ts file. This file contains all the mock data for casino games, promotions, and news articles. You do not need a database for this assignment; all content will be served directly from this file.

II. Core Requirements (Mandatory)

A. Backend (Express with TypeScript)

1. Project Setup: Initialize a new project and configure it to use TypeScript.
2. Express Server: Create a simple Express.js server that listens on a specified port (e.g., 5000).
3. CORS Handling: Configure CORS (Cross-Origin Resource Sharing) to allow your frontend application (which will run on a different port, e.g., 3000) to fetch data from the backend.
4. API Endpoint: Create a single GET endpoint, for example, /api/content, that returns the entire casinoContent object from the provided casinoContent.ts file as JSON.
   * Example: A request to GET http://localhost:5000/api/content should return the full JSON data.

B. Frontend (ReactJS with TypeScript)

1. Project Setup: Initialize a new React project with TypeScript.
2. Data Fetching: Fetch the content from your backend API (/api/content) once when the React application loads. Manage loading states (e.g., show "Loading..." while fetching).
3. Main Layout & Navigation:
   * Create a basic application layout with a header.
   * Implement a simple navigation (e.g., using buttons, tabs or basic links) to switch between displaying "Casino Games," "Promotions," and "News."
   * Only the content for the currently selected category should be visible in the main content area.
4. Casino Games Display:
   * When "Casino Games" is selected, display a responsive grid of all casino games from the fetched data.
   * Each game should be presented as a "card" showing its title, provider, and imageUrl.
   * Apply some basic CSS to make the game cards visually distinct and the grid layout organized.
5. Client-Side Filtering & Search for Casino Games:
   * Implement a search bar that allows users to filter the displayed casino games by title (case-insensitive, partial matches).
   * Implement filter buttons or a dropdown that allow users to filter games by one or more categories (e.g., "slots", "live-casino", "table-games", "jackpot").
   * The filtering/search should work in combination (e.g., searching for "star" within "slots" category).
   * The displayed games should update dynamically as the user types or applies filters.
6. Promotions Display:
   * When "Promotions" is selected, display a list of promotion cards. Each card should show its title and a snippet.
   * Implement a "Read More" mechanism (e.g., a button that expands the card to reveal fullTerms, or a simple modal/popup) for each promotion.
7. News Display:
   * When "News" is selected, display a list of news articles. Each item should show its title and date.
   * Implement a "Read Article" mechanism (similar to promotions, expanding the item or showing a modal) to display the fullContent.
8. Basic Styling: Apply sufficient CSS to make the application presentable, readable, and demonstrate an understanding of responsive design (e.g., ensuring cards adapt to different screen sizes).

C. Testing (Basic Unit Tests)

1. Backend Unit Test (1): Write at least one simple unit test for a backend utility function or part of your API logic.
2. Frontend Unit Test (1): Write at least one simple unit test for a React component or a client-side utility function.

III. Deliverables

1. Git Repository: Please create a public Git repository (on GitHub, GitLab, or Bitbucket) for your solution.
2. README.md File: Include a clear and concise README.md at the root of your repository with:
   * Project Overview: A brief description of the application.
   * Setup Instructions: Step-by-step instructions on how to clone the repository, install dependencies for both frontend and backend, and run the application locally.
   * Running Tests: Instructions on how to run the unit tests.
   * Key Decisions: Briefly describe any significant design or architectural choices you made given the time constraint.
   * Challenges & Next Steps: Mention any challenges you faced and how you approached them. If you ran out of time, briefly explain what you would implement or improve next.

IV. Evaluation Criteria

Your submission will be evaluated based on the following:

1. Functionality (40%): Does the application meet all mandatory requirements? Is it fully functional, including data fetching, navigation, content display, and client-side filtering/search?
2. Code Quality & Best Practices (30%):
   * Readability & Maintainability: Is the code clean, well-organized, and easy to understand?
   * Modularity: Is the code broken down into logical components/modules?
   * TypeScript Usage: Effective and appropriate use of TypeScript for type safety.
   * React Best Practices: Proper use of components, state, props, and hooks.
   * Express Best Practices: Efficient API design, proper error handling (even simple ones).
   * HTML/CSS: Semantic HTML, and a functional, readable UI.
3. Problem Solving & Time Management (20%): How effectively did you prioritize and deliver a working solution within the tight timeframe?
4. Testing & Documentation (10%): Quality of basic unit tests and clarity/completeness of the README.md file.

Good luck! We look forward to seeing your work and how you approach building a clean and functional application under a realistic time constraint.