Take Home Assessment Overview

This technical assessment is for a senior C# .NET Core developer role. It emphasizes depth in designing and implementing a backend API, focusing on how you approach architecture, code quality, security, and best practices. Rather than a broad application, prioritize a small number of well-implemented endpoints (e.g., 2-3) that demonstrate thoughtful decision-making.

The theme is a **"Dungeon Explorer" API** – a scenario involving a grid-based dungeon map with a hero starting point, obstacles (walls), and a treasure goal. The core challenge is to handle map creation and path computation in a way that requires logical problem-solving, such as determining a route from start to goal. This allows room for creativity: you could implement a proper pathfinding algorithm (e.g., A* for efficiency on grids), but it's optional – a simplified or hardcoded approach is acceptable if it fits your design.

The backend must be a .NET Core Web API, with data persistence (e.g., via a database). Include a minimal frontend SPA using Aurelia, React, or Angular to interact with the API. The entire solution should be containerized (e.g., using Docker) and include unit tests for key components.

Deadline: 7 days from receipt of assignment

Tech Stack Guidelines:

- Backend: .NET Core (C#), ASP.NET Core for the API.
- Database: Use Entity Framework Core with a suitable relational DB (choices like SQLite or PostgreSQL are up to you).
- Frontend: Select Aurelia, React, or Angular; keep it simple for API demonstration.
- Containerization: Docker-based, with a setup that allows easy running (e.g., including multi-container support if needed).
- Testing: Include unit tests using a framework like xUnit or NUnit.
- Other: Incorporate tools like Swagger if it aids your design; focus on configuration, logging, and error handling as you see fit.

Evaluation Criteria:

- **Architecture:** Demonstrate patterns like dependency injection, layered structure, or repositories – choose what best suits the problem.
- **Logical Challenge:** Show problem-solving in path computation; optional to implement a full algorithm (e.g., A*), but it highlights strong candidates.
- **Security and Best Practices:** Address concerns like input validation, secure data handling, and mitigations for common vulnerabilities (e.g., OWASP Top 10 elements such as SQL injection prevention).
- **Testing:** Cover critical logic, services, and edge cases.
- **Containerization:** Ensure the app runs reliably in containers.
- **Code Quality:** Readable, maintainable code with appropriate handling of errors and performance.

- **Depth Over Breadth:** Focus on polished, thoughtful implementations rather than quantity.
- **Decision-Making:** We'll evaluate choices like HTTP methods, data models, and optimizations without prescribing them.

Project Requirements

1. Backend API (Primary Focus)

Design a RESTful API to manage dungeon maps and compute paths. Aim for a minimal set of endpoints that cover:

- Creating and storing a dungeon map (e.g., grid dimensions, start position, goal, obstacles).
- Retrieving a map and computing a path from start to goal, avoiding obstacles.

Example data structure (adapt as needed):

```
{
  "width": 10,  // Grid width (int, 5-50)
  "height": 10, // Grid height (int, 5-50)
  "start": { "x": 0, "y": 0 }, // Hero start position
  "goal": { "x": 9, "y": 9 },  // Treasure position
  "obstacles": [ { "x": 1, "y": 1 }, { "x": 2, "y": 3 } ] // Array of wall positions
}
```

For path computation(optional):

- Implement logic to find a valid route. You can keep it simple (e.g., a hardcoded path for testing) or opt for a more robust algorithm like A* (using a priority queue and heuristic like Manhattan distance). The choice is yours – it should align with efficient, logical design.
- Handle cases like invalid maps, no path available, or larger grids.

Incorporate persistence to store maps (e.g., in a DB table). Use best practices for API design, such as appropriate HTTP methods, response formats, and validation. Secure the implementation against common risks.

2. Frontend Integration

Build a basic SPA to interact with your API:

- Allow input of dungeon details (e.g., via forms or interactive grid).
- Submit to create a map.
- Display the map and computed path (e.g., visually on a grid). Use your chosen framework for API calls (e.g., via HTTP client libraries). Keep the UI straightforward to emphasize backend integration.

3. Containerization

Package the solution in Docker containers. Include necessary files (e.g., Dockerfile, compose setup) for building and running the app, handling dependencies like the database.

4. Unit Tests

Add tests for core functionality, such as API logic, validation, and path computation. Focus on isolation and coverage of key scenarios.

5. Optional Enhancements (If Time Allows)

- Add features like authentication, random map generation, or extended error details.
- Improve visualization or API documentation.

Setup and Submission Instructions

1. **Environment:** Use your preferred tools (e.g., Visual Studio, .NET SDK, Node.js for frontend, Docker).
2. **Submission:** Provide a zipped repo or GitHub link, including a README with setup/run instructions (e.g., Docker commands, how to test endpoints).
3. **Testing:** We'll run your solution, interact via tools like Postman, review code, and execute tests.