Full Stack Engineer Technical Assessment

Objective

This assessment evaluates your ability to design and implement a well-architected C# .NET API with emphasis on clean architecture, database design, and your ability to explain technical decisions.

Instructions

1. You may use any online resources, documentation, or tools you typically use in your day-to-day work.
2. **Prioritize depth of understanding over feature completeness.** We want to see how you think and make architectural decisions.
3. **Target time: ~1-2 hours. It's ok not to finish – but be prepared to discuss challenges and what/how you would complete the project if you had more time!**
4. Use free database options only (SQLite, SQL Server Express LocalDB, or PostgreSQL via Docker) - your choice won't impact scoring
5. Upon completion, provide a link to your GitHub repository with a detailed README.md
6. Create a public GitHub repository (do not use "RegScale" in the repository name or code)
7. **You will present your solution live**, walking us through your code, explaining your decisions, and discussing trade-offs

Part 1: C# .NET Web API Backend (Primary Focus)

Database Schema

Design a relational database with:

Product: Id, Name, Description, Price, CategoryId (FK), StockQuantity, CreatedDate, IsActive

Category: Id, Name, Description, IsActive

Requirements:

- Proper foreign key relationship (Product → Category)
- Add indexes to support your query patterns
- Document your indexing decisions in README

API Endpoints

Products:

- GET /api/products - All active products with category info
- GET /api/products/{id} - Specific product (404 if not found or inactive)
- POST /api/products - Create product

- PUT /api/products/{id} - Update product (404 if not found or inactive)
- DELETE /api/products/{id} - Soft delete (set IsActive = false)

Categories:

- GET /api/categories - All active categories
- POST /api/categories - Create category

Complex Endpoint (Choose ONE):

Option A - Product Search:

```
GET
/api/products/search?searchTerm=&categoryId=&minPrice=&maxPrice=&inStock=&sor
tBy=&sortOrder=&pageNumber=&pageSize=
```

- All parameters optional and combinable
- Search: case-insensitive "contains" across name and description (AND logic for multiple words)
- Returns: { items: [], totalCount, pageNumber, pageSize, totalPages }
- Only active products
- Efficient single-query implementation

Option B - Category Analytics:

```
GET /api/categories/{id}/summary
```

Returns:

```
{ "categoryId": 1, "categoryName": "Electronics", "categoryDescription":
"...", "totalProducts": 25, "activeProducts": 23, "averagePrice": 149.99,
"totalInventoryValue": 34497.75, "priceRange": { "min": 9.99, "max": 999.99
}, "outOfStockCount": 5 }
```

- totalInventoryValue = sum of (Price × StockQuantity) for all products
- outOfStockCount = count of products where StockQuantity = 0
- 404 if category doesn't exist
- Efficient aggregation query

Architecture Requirements

1. **Entity Framework Core**
    - Use EF Core as primary ORM
    - Use .AsNoTracking() for read-only operations
    - Avoid N+1 queries (eager loading, projections)
    - Use raw SQL only with clear performance justification
2. **Clean Architecture**
    - Service layer for business logic (thin controllers)
    - Repository pattern OR direct DbContext (your choice - explain trade-offs)

    - Dependency injection
    - Single Responsibility and Dependency Inversion principles
3. **DTOs & Validation**
    - Separate DTOs from entities
    - Input validation (required fields, price > 0, stock >= 0)
4. **Error Handling**
    - Global exception handling
    - Appropriate HTTP status codes (200, 201, 204, 400, 404, 500)
    - Consistent error response format
5. **Seed Data**
    - 4-5 categories, 15-20 products
    - Varied prices and stock levels
    - Include in repository

Part 2: Angular Frontend (Minimal)

Minimum Implementation

Product List View:

- Fetch and display products from API
- Show: Name, Price, Category Name, Stock Quantity
- Use Angular service with DI
- Use *ngFor for display
- Display error message on API failure

Technical Requirements:

- Current version of Angular
- Angular service for HTTP calls
- Dependency injection
- **Basic routing** (at minimum: route to product list view)
- *ngFor and *ngIf directives

Styling: Not evaluated - unstyled HTML is fine

Optional (if time permits): Product form, filters, search, details view, additional routes

Part 3: Documentation

README.md Requirements:

1. **Quick Start**
    - Prerequisites
    - Setup and run instructions
2. **Architecture**

    - Overall architecture approach
    - Database schema
    - Technology choices
3. **Design Decisions**
    - How you applied Single Responsibility and Dependency Inversion
    - EF Core approach and query optimization
    - Complex endpoint choice and rationale
    - Repository pattern decision and trade-offs
    - Index strategy
4. **What I Would Do With More Time**
    - Unimplemented features and approach
    - Refactoring priorities
    - Production considerations
5. **Assumptions & Trade-offs**
    - Key assumptions made
    - Trade-offs in your design

Part 4: Live Presentation

You will walk the team through:

1. Code structure and key components (~15 min)
2. Design decisions and trade-offs (~10 min)
3. Q&A on alternative approaches, scaling, extensibility (~15-20 min)

We value: Clear reasoning, architectural thinking, ability to discuss trade-offs

Prioritization Guidance

If short on time, prioritize:

1. Basic CRUD working with clean architecture
2. Complex endpoint implemented
3. Clear documentation of decisions
4. Basic frontend functionality

Document what you didn't finish - we evaluate your thinking as much as completed features.

Submission

Public GitHub repository with code, README, and seed data. Do not use "RegScale" in repository name or code.