Scott Eltringham

TECHNICAL EXERCISE

Lead Software Engineer

[Introduction 2](#_Toc132773139)

[Project Overview 3](#_Toc248889141)

[Development Process 3](#_Toc1078856312)

[Requirements Gathering 4](#_Toc2133477732)

[Design 4](#_Toc1935972433)

[Key Components 4](#_Toc1449834072)

[Diagrams 4](#_Toc529240577)

[Implementation 6](#_Toc1797597533)

[Frameworks/Libraries 6](#_Toc1263431638)

[Code Examples 6](#_Toc1955303249)

[Testing 6](#_Toc838563628)

[Engineering Practices 7](#_Toc1969358368)

[Coding Standards 8](#_Toc503704519)

[Naming Conventions & Code Formatting 8](#_Toc124610844)

[Design Patterns 8](#_Toc1712292893)

[Delivering Confidence to Users 8](#_Toc364137315)

[Transparency 8](#_Toc1724593291)

[Reliability 9](#_Toc1309961403)

[User Feedback 9](#_Toc1237682109)

[Security 9](#_Toc1287770175)

[Time Allocation 10](#_Toc406961231)

## **Introduction**

In this document, I present my technical exercise submission, aiming to showcase my engineering skills and demonstrate my software solution. This exercise is designed to highlight my ability to tackle complex technical challenges and demonstrate the quality and reliability of my software solutions. By following best practices and maintaining high standards of engineering excellence, I ensure that my solutions are effective and dependable, meeting the needs of stakeholders and users.

### Project Overview

The deliverable of this project is to build a new product that allows users to plan and review their meals by tracking their calories with a limit of 1800 calories per day. To achieve this, the objective is to apply Engineering Excellence practices, the processes that go into creating the product, thought processes throughout the development and how IS give confidence to the users in the software products I build.

## Development Process

### **Requirements Gathering**

To gather the requirements for this project, I began by researching the key objectives and goals outlined in the project brief. In the absence of explicit user input for this exercise, I made informed assumptions based on similar products and industry best practices.

In a typical project, I would gather feedback from stakeholders and the product team to refine the requirements. However, since the project brief was my only source of requirements, I relied on my own experience and insights to guide the development process.

### **Design**

The design of this application follows a vertically sliced architecture, emphasizing functionality over strict layering. This is done to avoid polluting the business logic code with concerns around the exact way the data is presented to users, or how the data persists, and these modules can change independently from each other. Each feature or use case is encapsulated within its own module, consisting of components from different layers that collaborate to achieve specific tasks.

#### **Key Components**

* **Blazor UI**: Serves as the front-end interface for users to interact with the application. It consists of various components and pages built using C# and Razor syntax.
* **Endpoints**: These represent the entry points for various operations within the application. Each endpoint encapsulates a specific action or use case, such as creating a meal or logging in.
* **Services**: Services contain the business logic required to fulfill the functionality defined by the endpoints. They coordinate the interactions between different components and enforce business rules.
* **Repositories**: Repositories handle data access and storage. They provide an abstraction layer over the underlying data source and facilitate CRUD operations on entities.
* **Entities**: Entities represent the core data objects manipulated by the application. They encapsulate the business entities such as meals and accounts.

#### **Diagrams**

Given the vertically sliced nature of the architecture, the traditional UML diagrams like class diagrams and sequence diagrams may not fully capture the structure. Instead, each feature or functionality can be visualized as a standalone slice with its own components and interactions.

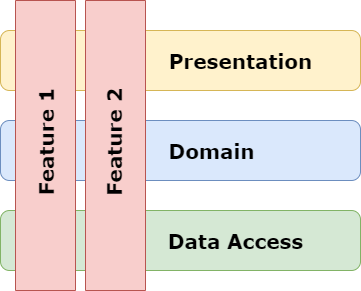


Figure 1. An example of vertically sliced features

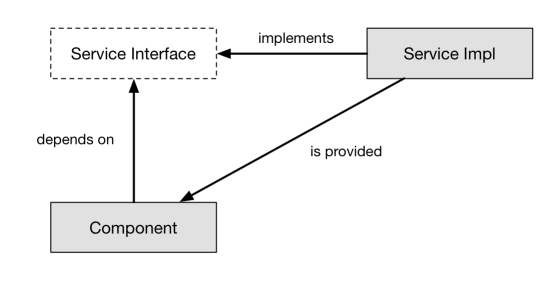


Figure 2. Typical example of dependency injection via Interfaces

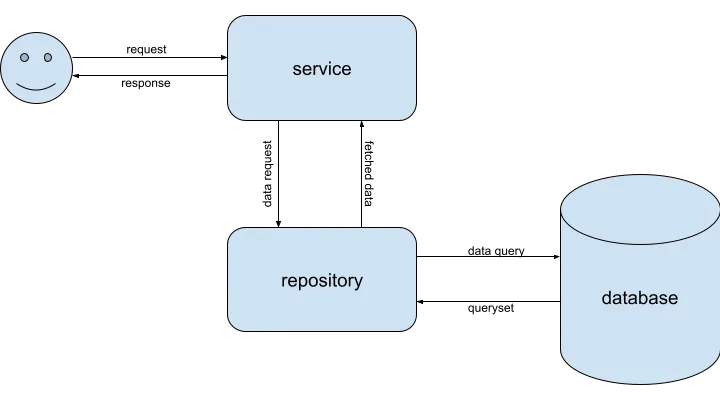


Figure 3. A service-repository pattern is a low-level design pattern mainly used to separate the concerns in a software project.

### **Implementation**

#### Frameworks/Libraries

* **ASP.Net Core**: A cross-platform, high-performance, open-source framework for building modern, cloud-enabled, Internet-connected apps.
* **Entity Framework Core**: A popular ORM (Object-Relational Mapper) for .NET applications.
* **Fluent Validation**: Library - A popular library for fluent validation in .NET applications.
* **Blazor**: A modern front-end web framework based on HTML, CSS, and C#

#### **Code Examples**

Here are some key code snippets illustrating the design approach:

1. **Endpoints and Services**:
   * MealCreateEndpoint: Represents an endpoint for creating a new meal. It delegates the actual creation logic to the MealCreateService.
   * MealCreateService: Contains the business logic for creating a new meal. It interacts with the data layer through a repository.
2. **Repositories**:
   * CreateRepository, ReadRepository, UpdateRepository, DeleteRepository: Provide generic implementations for CRUD operations on entities. They are used by services to interact with the underlying data store.
3. **Blazor Component**:
   * The login page demonstrates the UI component responsible for user authentication. It interacts with the authentication service to handle user login.

### **Testing**

To ensure the quality and reliability of the application, various testing strategies were employed, utilizing specific frameworks and tools for unit, integration, and end-to-end testing.

* **Unit Testing - Framework**: MSTest
  + **Description**: MSTest is a robust testing framework for .NET applications that offers a wide range of assertions and supports data-driven testing.
  + **Example**: Unit tests were written to verify the functionality of individual methods and classes, ensuring they produce the expected outcomes under various conditions. For example, tests were created to validate the business logic in the MealCreateService to ensure it correctly calculates calorie totals and handles edge cases.
* **Integration Testing - Framework**: xUnit
  + **Description**: Integration tests were conducted using xUnit. This setup allows for testing the interactions between different components of the application in a realistic environment.
  + **Example**: Tests were created to verify that different services and repositories work together correctly, ensuring data is correctly processed and persisted. For instance, tests were performed to check when the MealCreateEndpoint is called, the MealCreateService correctly interacts with the CreateRepository to save new meal entries to the database.
* **Manual Testing**:
  + **Approach**: In addition to automated testing, thorough manual testing was conducted to catch edge cases and ensure a smooth user experience. This involved exploratory testing and usability testing to identify any issues not covered by automated tests.
  + **Example**: Manual tests included navigating through the UI to ensure all elements functioned correctly and that the overall user experience was intuitive and error-free.

By leveraging these frameworks and tools, we were able to perform comprehensive testing at various levels, ensuring the application is robust, reliable, and ready for production.

## Engineering Practices

### **Coding Standards**

Adhering to consistent naming conventions and code formatting is crucial for maintaining code readability and maintaining a clean and organized codebase.

#### **Naming Conventions & Code Formatting**

The following conventions and practices were followed throughout the project:

* **Classes and Interfaces**: Pascal Case (e.g., MealCreateService, ICreateRepository)
* **Methods**: Pascal Case (e.g., AddAsync, DeleteAsync)
* **Variables and Parameters**: Camel Case (e.g., meal, context)
* **Namespaces**: Pascal Case, matching the directory structure (e.g., CalorieWise.Api.Services)
* **File Structure**: One class per file, with the file name matching the class name.

### **Design Patterns**

To ensure maintainability, scalability, and a clear separation of concerns, several design patterns were employed:

* **Vertical Slice Pattern:** Technique that helps us build maintainable applications by separating the application around features.
* **Repository Pattern**: Used in the Data Layer to abstract data access logic. It provides a flexible way to manage data operations.
  + *CreateRepository*: Handles the addition of new entities.
  + *DeleteRepository*: Manages the deletion of entities by their identifier.
  + *ReadRepository*: Facilitates querying and retrieving entities.
  + *UpdateRepository*: Manages updating existing entities.
* **Dependency Injection (DI)**: Implemented to inject dependencies into classes, improving testability and modularity.

### **Delivering Confidence to Users**

In a real-world project, it's crucial to follow best practices to ensure user confidence. While some of these practices couldn't be fully implemented due to the nature of this tech test, here are the key principles we strive to adhere to.

#### Transparency

Transparency is essential for building trust with users. We ensure users are kept informed about the development progress and updates through regular communication channels. This includes:

* **Release Notes**: Detailed documentation of new features, bug fixes, and improvements in each release.
* **Progress Updates**: Periodic updates via newsletters, blogs, or in-app notifications to keep users aware of ongoing development efforts.
* **Roadmaps**: Sharing plans and milestones to set clear expectations about upcoming features and enhancements.

#### **Reliability**

To provide a reliable product, we prioritize rigorous testing and continuous monitoring:

* **Automated Testing**: Implementing comprehensive automated test suites, including unit tests, integration tests, and end-to-end tests, to ensure functionality and stability.
* **Manual Testing**: Conducting thorough manual testing, especially for critical paths and new features, to catch any edge cases or user experience issues.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Utilizing CI/CD pipelines to automate the testing and deployment process, ensuring that only thoroughly vetted code reaches production.
* **Monitoring**: Employing real-time monitoring and alerting systems to promptly detect and address issues, ensuring minimal downtime and quick resolution of any problems.

#### **User Feedback**

Actively soliciting and incorporating user feedback is vital for creating a product that meets users' needs and expectations:

* **User Surveys**: Regularly conducting surveys to gather user opinions, satisfaction levels, and suggestions for improvements.
* **Feedback Channels**: Providing easy-to-access feedback mechanisms within the application, such as feedback forms, support tickets, and user forums.
* **User Testing**: Engaging users in beta testing and usability studies to gather direct feedback on new features and identify areas for improvement.
* **Iterative Development**: Adopting an agile development approach, where user feedback is continuously integrated into the development cycle, ensuring the product evolves in line with user needs.

#### **Security**

Implementing best practices for data security and privacy is non-negotiable to protect user data and build trust:

* **Data Encryption**: Ensuring all sensitive data is encrypted both in transit and at rest to prevent unauthorized access.
* **Authentication and Authorization**: Implementing robust authentication mechanisms (e.g., multi-factor authentication) and fine-grained authorization controls to secure user access.
* **Security Audits**: Conducting regular security audits and vulnerability assessments to identify and mitigate potential security risks.
* **Compliance**: Adhering to industry standards and regulations (e.g., GDPR, HIPAA) to ensure data privacy and security compliance.
* **Incident Response**: Establishing a clear incident response plan to swiftly address any security breaches or data leaks, minimizing impact and restoring user confidence.

## Time Allocation

Below is a breakdown of the time spent on each phase of the project, along with explanations of any significant challenges or considerations that influenced the time allocation. The implementation phase required more time due to the complexity of integrating different components and ensuring seamless functionality. Testing was also a significant focus to ensure the reliability of the final product.

|  |  |  |
| --- | --- | --- |
| **Phase** | **Time Spent (hours)** | **Description** |
| **Requirements Gathering** | 2 | Focused on interpreting the project brief and making informed assumptions due to the absence of explicit user input. |
| **Design** | 4 | Spent designing the application's architecture, ensuring a robust and maintainable structure. |
| **Implementation** | 27 | Much of the time was dedicated to coding the application, implementing key features, and ensuring functionality. This phase also involved integrating various frameworks and libraries. |
| **Testing** | 12 | Conducted unit, integration, and end-to-end tests to ensure the application’s reliability and performance. This phase was crucial for identifying and resolving any issues. |
| **Documentation** | 3 | Prepared detailed documentation to explain the development process, design decisions, and usage of the application. Ensured clarity and completeness to aid future maintenance and user understanding. |