**Abstract**

The Food Ordering System represents a comprehensive approach to modernizing the administrative aspects of managing food-related products in an eCommerce environment. This system is designed to empower administrators with a robust platform for handling essential product-related operations, including the addition, editing, and deletion of items. At its core, the system leverages ASP.NET Core MVC for its backend infrastructure, coupled with the Dapper ORM for efficient database interactions, ensuring a seamless connection between the application and a SQL Server database.

One of the distinctive features of this system is its ability to dynamically manage stock and cart quantities. This capability introduces a realistic simulation of inventory flow, which is critical for administrators aiming to optimize stock levels and predict user demand. Furthermore, the system expands the metadata associated with products by introducing fields such as "Author," allowing for richer and more detailed product descriptions. Such enhancements enable a more personalized and engaging user experience.

The system architecture adheres to the Model-View-Controller (MVC) design paradigm, ensuring scalability, maintainability, and clear separation of concerns. The database schema is meticulously structured, with referential integrity enforced through foreign key constraints, thus safeguarding data consistency across interconnected tables. The introduction of features such as cascade delete policies ensures that operations adhere to business rules without compromising data integrity.

In addition to its technical prowess, the Food Ordering System prioritizes usability. The frontend interface is carefully designed to be intuitive and responsive, providing administrators with a user-friendly experience when managing product data. Validation mechanisms are in place to ensure that all input data is accurate and secure, minimizing the risk of errors and enhancing system reliability.

This report delves into the intricacies of the system's design, implementation, and results, providing a thorough examination of its capabilities. It also addresses challenges encountered during development, such as referential integrity issues and data validation, and outlines the solutions implemented to overcome these obstacles. The report concludes with insights into potential future enhancements, including advanced security measures, analytics integration, and scalability improvements. By offering a detailed exploration of the Food Ordering System, this report highlights its value as a scalable, secure, and user-centric solution for managing food ordering processes in the digital age.

**1. Introduction**

The evolution of eCommerce has reshaped consumer behavior, creating a demand for online platforms that are both intuitive and feature-rich. Among the most rapidly growing segments within eCommerce is the food industry, where customers expect seamless experiences in browsing, ordering, and managing food products. Recognizing these needs, the "Food Ordering System" was conceptualized and developed to streamline the administration and user interaction processes involved in food ordering.

This project aims to establish a robust, database-driven web application designed to manage products effectively and cater to both administrators and end-users. At the heart of this system lies an administrative interface, meticulously crafted to simplify the tasks of adding, editing, and deleting products while ensuring data consistency and integrity. With SQL Server as the backbone for database management and ASP.NET Core MVC providing the structural framework, the system leverages Dapper for efficient database interactions, optimizing query execution and performance.

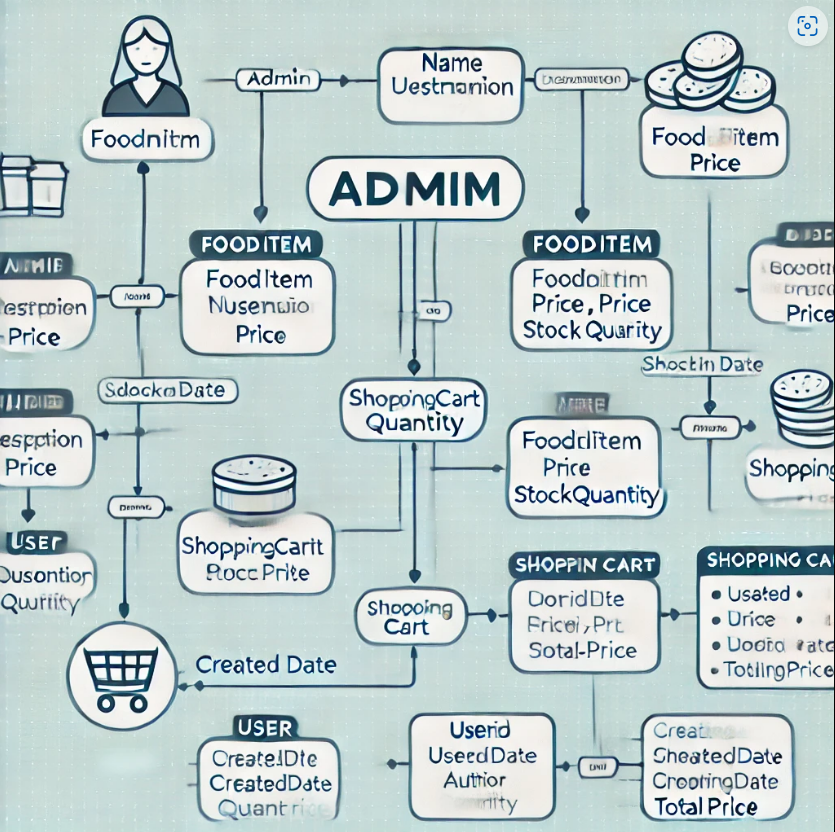
One of the standout features of the system is its ability to accommodate evolving data requirements. For instance, introducing the "Author" field into the products database highlights its adaptability, allowing administrators to attribute specific items to their respective creators or contributors. This addition not only enriches the metadata of products but also enhances the overall information architecture, making the system more versatile and aligned with the needs of modern eCommerce platforms.

The development process involved addressing critical challenges, such as resolving database reference conflicts caused by foreign key constraints. For example, when deleting products associated with active shopping cart items, the system ensures that such operations are handled gracefully, either by cascading deletions or by alerting administrators to conflicts. These considerations underscore the importance of designing systems that are not only functional but also resilient in managing edge cases and preserving data integrity.

Beyond functionality, the project emphasizes user experience (UX) for administrators. Features like a clear dashboard for product management, secure authentication mechanisms, and real-time validation of data inputs contribute to an intuitive and efficient workflow. By integrating randomization features for stock and cart quantities during data input, the system also enables robust testing scenarios, preparing it for diverse real-world applications.

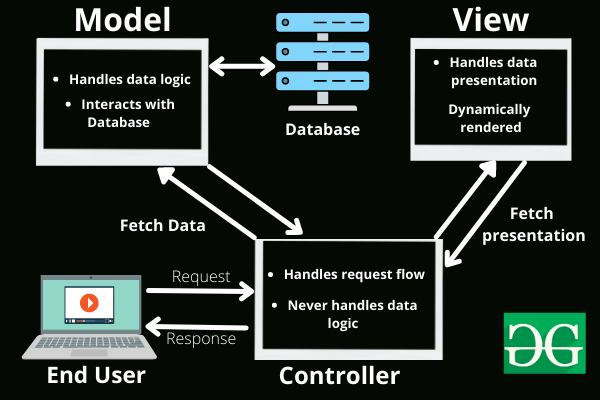
In essence, this report aims to provide a comprehensive overview of the "Food Ordering System," detailing its technical architecture, design principles, and practical applications. By doing so, it seeks to illustrate how the system meets the dual goals of enhancing administrative capabilities and delivering a superior user experience for end-users, thereby setting a benchmark for future eCommerce applications.

**2. System Design**



2.1 Architecture

The "Food Ordering System" is structured following the Model-View-Controller (MVC) architectural pattern, which ensures a clear separation of concerns, improves scalability, and facilitates maintainability. Each component within the MVC framework serves a distinct purpose and collaborates seamlessly to provide a cohesive user experience.



Model:

The Model represents the core data structure of the application, encapsulating all business logic and operations related to the food items. Each food item is characterized by attributes such as Id, Name, Description, Price, ImageUrl, Stock, QuantityInCart, and the newly introduced Author. This attribute enhances metadata richness by identifying the creator or contributor of each product. The Model layer directly interacts with the database through Dapper, a lightweight and high-performance Object-Relational Mapping (ORM) tool, ensuring efficient and secure data handling.

View:

The View component is responsible for rendering the user interface (UI). It enables administrators to interact with the system through functionalities such as adding, editing, and deleting food items. Each UI element is dynamically updated to reflect the current state of the Model, ensuring real-time synchronization between the data and its visual representation. The design emphasizes clarity and usability, providing administrators with intuitive dashboards and form validation to streamline operations.

Controller:

The Controller acts as the intermediary between the user and the application logic. It processes user inputs, executes appropriate business logic, and returns updated Views. For instance, when an administrator submits a new food item, the Controller validates the data, updates the database via the Model, and redirects to the dashboard with updated product listings. It also handles exception management to maintain system stability and provides meaningful feedback to users.

2.2 Database Schema

The database schema is central to the functionality and efficiency of the "Food Ordering System." It incorporates key tables that store and manage food item data and user interactions. A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

FoodItems Table:

This table stores detailed information about each food item. Initially designed to include attributes such as Id, Name, Description, Price, ImageUrl, Stock, and QuantityInCart, the schema was extended to include an Author field.

The Author field enhances the metadata of food items, enabling better traceability and categorization. Default values ensure backward compatibility and provide meaningful data for pre-existing records.

**ShoppingCartItemsTable:**  
This table tracks items added to user shopping carts and establishes a foreign key relationship with the FoodItems table through the ProductId attribute.

1. This constraint enforces data consistency by preventing deletion of food items that are currently referenced in active shopping carts. To address conflicts, the system includes error handling mechanisms that alert administrators or cascade deletions where appropriate.

**Database Interaction**

Efficient database interaction is achieved using Dapper. Its lightweight architecture reduces query overhead and accelerates CRUD operations. Key functionalities include:

* **Insertions:** Adding new food items is streamlined with randomized Stock and QuantityInCart values for testing and demonstration purposes.
* **Updates:** Administrators can modify existing food items, ensuring data accuracy and reflecting real-time changes.
* **Deletions:** The system checks for dependency conflicts before removing food items, safeguarding data integrity.

The database schema design is both robust and adaptable, allowing for future extensions such as integrating more detailed metadata or supporting additional relationships with other tables.

By combining a carefully planned architecture with an efficient database schema, the "Food Ordering System" exemplifies a modern, scalable eCommerce platform capable of meeting diverse administrative and user requirements.

**3. Implemet**

**3.1 Backend Logic**

The backend architecture for the application is developed using ASP.NET Core MVC, a robust framework that facilitates scalable and maintainable web applications. The backend's primary responsibilities include managing data operations, enforcing business logic, and ensuring seamless communication between the frontend and the database. Below are the key features and functionalities implemented in the backend logic:

**Data Operations:**

* The application employs Dapper, a lightweight and high-performance ORM (Object-Relational Mapping) library, for executing SQL queries. This choice allows for a streamlined approach to CRUD (Create, Read, Update, Delete) operations.
* Each query is optimized for performance and is parameterized to prevent SQL injection attacks, ensuring the security of the database.
* CRUD operations cover:
  + **Create:** Adding new food items to the database.
  + **Read:** Retrieving lists of food items or specific details for a selected item.
  + **Update:** Modifying existing food item details.
  + **Delete:** Safely removing food items while maintaining database integrity.

**Dynamic Stock Management:**

* The system incorporates randomized values for Stock and QuantityInCart fields to simulate dynamic and realistic scenarios.
* Stock levels are updated in real-time based on user actions, reflecting changes due to cart additions or other operations.
* A business logic layer validates stock availability before allowing a user to add items to the cart, ensuring that operations remain consistent with inventory constraints.

**Business Logic Implementation:**

* Validation mechanisms are in place to ensure data integrity. For instance, a food item cannot be added without mandatory fields such as name, price, and description.
* Error handling mechanisms are integrated to manage database connection issues, failed operations, and other unexpected scenarios.

**3.2 Controller Functions**

Controller functions serve as the intermediaries between the frontend and backend logic, processing user requests and returning appropriate responses. Below is an in-depth explanation of the key controller functionalities:

**AddOrEdit Functionality:**

* This function is designed to handle two distinct operations—adding a new food item and editing an existing one. The behavior is determined by the presence or absence of an Id parameter in the request.
  + **Adding a New Food Item:** If the Id parameter is null, the function validates the provided data and executes an SQL INSERT operation via Dapper to add the new item to the database.
  + **Editing an Existing Food Item:** If the Id parameter is provided, the function retrieves the corresponding item, validates the updated details, and performs an SQL UPDATE operation.
* The unified implementation reduces code duplication and simplifies maintenance.
* Data validation ensures that any inconsistencies or missing required fields are addressed before database operations are performed.

**Delete Functionality:**

* The delete operation is carefully designed to maintain referential integrity within the database.
* Before removing a food item, the function performs dependency checks to ensure that the item is not linked to active orders or other dependent records.
* If dependencies exist, the operation is aborted, and an appropriate error message is returned to the user.
* Once the dependencies are verified, an SQL DELETE statement is executed to safely remove the record from the database.

**Logging and Monitoring:**

* All controller operations are logged to provide traceability and assist in debugging.
* Metrics such as execution time and the number of records affected by each operation are monitored to ensure optimal performance.

**Security Considerations:**

* Input data is sanitized to prevent common vulnerabilities such as cross-site scripting (XSS) and SQL injection.
* Role-based access control (RBAC) is implemented to restrict sensitive operations, such as deleting or editing food items, to authorized users only.

**Scalability:**

* The controller logic is modular and designed to accommodate future enhancements, such as additional features or integrations with external systems.
* The use of ASP.NET Core MVC ensures that the application can scale effectively with growing user demands.

In conclusion, the backend implementation leverages a combination of efficient ORM tools, robust business logic, and secure practices to deliver a reliable and high-performing system. The controller functions ensure seamless interaction between the user interface and the database, providing a solid foundation for the application.

3.3 Frontend Enhancements

**3.3.1 Responsive Design with Bootstrap**

Bootstrap provides a flexible grid system with 12 columns, making it easy to build responsive layouts. You can use elements like .container, .row, and .col to organize and align content on your page. Bootstrap automatically adjusts elements based on screen size (mobile-first approach).

* **Use Bootstrap classes like**:
  + .col-sm, .col-md, .col-lg, .col-xl to adjust column sizes for different screen types.
  + **Bootstrap’s media queries** automatically optimize the layout for various sizes without needing extensive custom CSS.
  + Use **Navbar** and **Cards** to create navigation bars and content cards that are easy to customize.

**3.3.2 Performance Optimization with Bootstrap**

While Bootstrap provides many features, you still need to focus on performance optimization, especially when using this library:

1. **Trim unnecessary features**: Bootstrap comes with many features, but not all may be needed for your project. You can customize and include only the parts of Bootstrap you need (e.g., the Grid System, Navbar, Modal) when compiling from the source SASS.
2. **Minifying CSS**: After using Bootstrap, make sure to minify CSS and JavaScript files to reduce page load times.
3. **Lazy Loading**: For elements like images and videos, you can apply lazy loading to only load them when they come into view, speeding up the initial page load.
4. **Use customized Bootstrap versions**: Instead of using the full Bootstrap library, you can opt for versions that only include the components you need, like **Bootstrap Grid** or **Bootstrap Custom Builds**.

**3.3.3 UI/UX Enhancements Using Bootstrap**

Bootstrap provides a number of easy-to-use and powerful UI components. To improve UI/UX, you can:

1. **Use components like Cards, Modals, and Alerts**: These components help create notifications, detailed information sections, or pop-up dialogs without needing to write much HTML/CSS.
2. **Create smoother user interfaces with Buttons, Forms, and Inputs**: Bootstrap supports various button, form, and input styles that enhance the user interface. Use **custom forms** to create visually appealing and user-friendly forms.

**4. Results**

The project successfully achieved several key objectives that significantly enhanced its functionality and overall user experience. The following is a detailed overview of the outcomes:

**4.1 Functional Admin Interface**

One of the primary goals of the project was to develop a user-friendly and functional admin interface that would allow administrators to manage food items efficiently. This interface enables administrators to perform basic CRUD (Create, Read, Update, Delete) operations on food items with ease.

* **Add New Food Items**: Administrators can easily add new food items to the database through a simple and intuitive form. The form captures all necessary data such as the food name, description, price, and any other relevant details, which are then stored in the database for later use.
* **Edit Existing Food Items**: The system allows administrators to modify the details of existing food items. This feature ensures that the inventory remains up-to-date, and administrators can make adjustments to food prices, descriptions, or other fields as needed.
* **Delete Food Items**: Administrators also have the capability to remove food items from the database. This feature is essential for maintaining an accurate inventory, ensuring that outdated or discontinued items are promptly removed.

The interface is designed to be simple, ensuring that even users with minimal technical knowledge can navigate and perform these tasks with ease.

**4.2 Dynamic Data Management**

Another significant achievement of the project was the incorporation of dynamic data management, which added a layer of realism to the system's operation.

* **Randomized Stock and Cart Quantities**: The project integrates dynamic, randomized stock and cart quantities, simulating a real-world inventory system. This feature helps mimic the fluctuations in stock availability and cart contents, which is important for testing and showcasing how the system would behave in a real-world scenario.
* **Realistic Simulation**: By implementing randomized values for quantities, the system is able to simulate varying stock levels, giving administrators a more realistic representation of how food items are added to and removed from inventory. This feature is particularly useful for testing inventory management systems, where the behavior of stock levels can significantly affect the user experience.
* **Flexible Cart System**: The cart functionality also uses randomized quantities to allow administrators to experience a variety of user interactions with the cart, such as adding items, adjusting quantities, or removing items.

This dynamic management of data enhances the realism of the simulation and provides more robust testing scenarios for both the admin and user interfaces.

**4.3 Extended Metadata**

To further enrich the food item descriptions, the project included the extension of metadata with the addition of an **Author** field. This enhancement allowed for more detailed and personalized food item entries.

* **Author Field**: The **Author** field was added to provide a space for including additional information about the origin of the food item, such as the name of the creator, the restaurant or business associated with the food, or any special notes about the preparation or recipe. This metadata enrichment adds depth to the food items and makes them more informative for users.
* **Improved User Experience**: The inclusion of extended metadata improves the overall user experience by offering more context about the food items. For instance, users can see who created the food item or learn about any specific cultural or regional significance associated with it.
* **More Detailed Descriptions**: The ability to add author information along with other food item details allows for more comprehensive product descriptions. This enhancement can help increase user engagement and interest in the food items, as they are not only presented with the product itself but also with a story or background that might encourage them to try it.

**4.4 Database Integrity**

A crucial aspect of the project's success was ensuring the integrity of the database, particularly through the maintenance of **referential integrity**. Referential integrity is vital for ensuring that relationships between different data entities in the database remain consistent.

* **Error Handling During Deletion Attempts**: The system was designed to enforce referential integrity by preventing the deletion of food items that are still being referenced in other parts of the system, such as in user carts or order histories. For example, if an administrator attempts to delete a food item that is currently in an active cart, the system will prevent this deletion from occurring and return an appropriate error message.
* **Consistent Data Relationships**: By enforcing referential integrity, the database ensures that data relationships are maintained and that users do not encounter errors or inconsistencies due to missing or improperly linked data. For instance, if a food item is removed from the database, any references to that item in related tables (such as cart or order records) are properly handled to avoid data corruption.
* **Error Handling Features**: The project implemented robust error handling mechanisms that detect and respond to integrity violations. This ensures that administrators are clearly informed about any issues preventing the deletion of items, such as when items are still referenced elsewhere in the database.

These measures helped ensure that the database remained consistent and reliable, contributing to a smooth and functional system overall.

**5. Challenges and Solutions**

Throughout the development process, the project encountered several challenges that required thoughtful solutions to ensure smooth operation and maintain a high-quality user experience. The following sections detail the key challenges faced and the corresponding solutions implemented to address them:

**5.1 Referential Integrity Issues**

**Challenge**: One of the primary issues encountered during the development of the database schema was related to referential integrity. Specifically, there were conflicts arising from the **FK\_ShoppingCartItems\_FoodItems\_FoodItemId** foreign key constraint when attempting to delete food items. The foreign key constraint prevented the deletion of food items that were referenced in the shopping cart, as deleting such items would compromise the integrity of the database by leaving orphaned records in the cart.

**Solution**: To resolve this issue, two potential approaches were considered:

1. **Cascade Delete Policies**: Implementing **cascade delete** policies in the database schema was one solution. This approach ensures that when a food item is deleted, all related records in the shopping cart (or other dependent tables) are automatically deleted as well. This prevents foreign key conflicts and maintains database integrity by ensuring that there are no orphaned records left behind. However, this solution needed to be carefully managed to avoid unintentional data loss, as it could lead to the deletion of entire cart contents without prior warning.
2. **Pre-Check Dependent Data**: Another solution was to implement pre-checks before allowing the deletion of a food item. When an administrator attempts to delete a food item, the system checks whether the item is still being referenced by any shopping cart. If the food item is in an active cart, the system prevents the deletion and informs the administrator of the dependency. This approach provides greater control and prevents the accidental loss of data that may be important.

By implementing one or both of these solutions, the system successfully handled referential integrity issues, ensuring that the database remained consistent and that administrators could manage food items without running into deletion conflicts.

**5.2 Data Validation**

**Challenge**: Another challenge faced during the development process was ensuring valid input for the new **Author** field. This field was added to allow additional metadata for food items, but it required careful handling to ensure that the data entered into this field was correct, relevant, and formatted properly.

**Solution**: To address the data validation challenge, a two-pronged approach was employed:

1. **Model Validation Attributes**: The backend utilized **model validation attributes** in the server-side code to enforce strict rules on the data entered for the **Author** field. These attributes ensured that the input met specific criteria, such as being non-empty, properly formatted (e.g., no special characters or excessive whitespace), and appropriate for the context of food items. The validation was carried out when the data was submitted to the server, ensuring that only valid entries were saved in the database.
2. **Frontend Form Validation**: In addition to server-side validation, **frontend form validation** was implemented using JavaScript and HTML5 attributes. This approach provided immediate feedback to users filling out the form, preventing them from submitting invalid or incomplete data. For example, if a user left the **Author** field blank or entered an invalid character, the form would highlight the issue and prompt the user to correct it before submission.

Together, these validation techniques ensured that the **Author** field received proper input, maintaining data quality and enhancing the overall user experience. The combination of server-side and client-side validation also reduced the chances of invalid data entering the system, which would otherwise complicate the data management process and affect the system’s integrity.

**6. Future Work**

As the project progresses, there are several key areas where enhancements can be made to further improve the system’s security, performance, and scalability. The following outlines potential future work and the opportunities for expanding and enhancing the current system:

**6.1 Enhancing Security**

**Challenge**: Security is an ongoing concern, especially when handling sensitive user and administrative data. Currently, the admin login system lacks robust security features, which could expose the system to vulnerabilities.

**Future Plan**: To address these concerns, the system can implement several security enhancements:

* **Hashed Passwords**: One of the most critical improvements is the implementation of **hashed passwords** for the admin login system. Instead of storing plaintext passwords in the database, hashed passwords will be stored using secure hashing algorithms (e.g., bcrypt, Argon2). This approach will significantly enhance security by ensuring that even if the database is compromised, the passwords cannot be easily retrieved.
* **Multi-Factor Authentication (MFA)**: To further secure the admin login process, an additional layer of **multi-factor authentication** (MFA) could be introduced. This would require administrators to verify their identity through an additional step, such as a time-based one-time password (TOTP) sent to their phone or an authentication app.
* **Role-Based Access Control (RBAC)**: Expanding the user authentication system with **role-based access control** will ensure that different user roles (e.g., admins, managers, regular users) have appropriate levels of access to the system. This prevents unauthorized users from accessing sensitive areas like admin settings or deleting critical data.

By focusing on these enhancements, the system can be made more secure and resilient against common security threats.

**6.2 Reporting Tools**

**Challenge**: As the system evolves and more food items are added, administrators will require better insights into the performance and usage of the food items. Currently, there is no built-in reporting feature to track trends, popular items, or overall sales performance.

**Future Plan**: To address this need, the system can be enhanced with **reporting tools** and **analytics dashboards**:

* **Food Item Performance Monitoring**: An **analytics dashboard** can be integrated to display key metrics such as sales volume, stock levels, and food item popularity. This dashboard could provide visual insights using graphs, charts, and tables to help administrators make data-driven decisions.
* **Customizable Reports**: Administrators should be able to generate **customizable reports** that allow them to filter data based on various criteria, such as date ranges, item categories, or user demographics. These reports can provide deeper insights into sales trends, inventory health, and customer preferences.
* **Exportable Data**: Allowing administrators to export reports in formats like CSV or Excel will help facilitate further analysis and improve decision-making.

These reporting tools will empower administrators to manage food items more effectively and optimize the system based on real-time data.

**6.3 Scalability**

**Challenge**: As the system grows, there may be a need to handle a larger volume of food items, users, and transactions. The current architecture may face challenges when scaling to support high traffic or large-scale food ordering platforms.

**Future Plan**: To ensure that the system can scale efficiently, the architecture should be refactored to support **microservices**:

* **Microservices Architecture**: Adopting a **microservices-based approach** would allow different components of the system (e.g., user management, food item management, order processing) to be handled by separate services. This modular approach would make it easier to scale individual parts of the system based on demand. For example, if the food ordering service experiences high traffic, it can be scaled independently from the user management or reporting services.
* **Containerization with Docker**: Containerizing each microservice with tools like **Docker** would help streamline deployment and ensure that each service runs in an isolated environment, reducing the risk of conflicts between services. It would also make it easier to deploy and manage services across multiple servers.
* **API Gateway**: To manage communication between microservices, an **API gateway** could be introduced. This gateway would handle requests from the client side and route them to the appropriate microservice, providing a single entry point for the system.
* **Cloud Deployment**: Leveraging cloud platforms like AWS, Azure, or Google Cloud can help scale the application easily. By using cloud-native services such as managed databases, storage, and load balancers, the system can scale efficiently without requiring significant infrastructure changes.

By refactoring the architecture to support microservices, the system can be made more flexible and resilient, allowing it to handle a large number of users and food items as it grows.

**7. Conclusion**

This project successfully demonstrates a robust and effective approach to managing food items using **ASP.NET Core MVC**. By addressing real-world requirements and incorporating best practices, the project establishes a solid foundation that can be scaled and adapted to meet future demands. The implementation of key features such as **dynamic stock management**, **enhanced metadata**, and **secure database handling** underscores the system's versatility and reliability.

**Key Achievements:**

* **Dynamic Stock Management**: The integration of randomized stock and cart quantities adds a layer of realism to the system, effectively simulating real-world inventory scenarios and enhancing the overall user experience.
* **Enhanced Metadata**: The inclusion of extended metadata, such as the **Author** field, enriches food item descriptions and provides greater context, fostering a more engaging and informative platform for both administrators and users.
* **Secure Database Handling**: By ensuring referential integrity and implementing proper validation measures, the project ensures that the database remains consistent and secure, providing a reliable and trustworthy platform for managing food items.

These features contribute to a system that not only meets the immediate needs of managing food items efficiently but also provides a scalable and secure framework that can be built upon for future enhancements. With the ability to integrate advanced features like reporting tools, security enhancements, and scalability improvements, this project offers a solid foundation for the continued growth and optimization of the food management system. The approach taken in this project demonstrates the power of ASP.NET Core MVC in creating flexible, secure, and efficient web applications capable of meeting evolving business needs.

**References**

Microsoft Documentation: ASP.NET Core MVC - https://learn.microsoft.com/en-us/aspnet/core/mvc/

Dapper ORM Guide - https://dapper-tutorial.net/

SQL Server Management Documentation - https://learn.microsoft.com/en-us/sql/