Code Refactoring and Enhancements

Backend

Program.cs

- Database Context Configuration:Configured ApplicationDBContext to use SQL Server with a connection string from the configuration. Ensures that the application can perform CRUD operations on the database using a reliable ORM, improving maintainability and scalability.

```
services.AddDbContext<ApplicationDBContext>(options =>
options.UseSqlServer(Configuration.GetConnectionString("Def
aultConnection")))
```

- CORS Policy:Added a CORS policy named AllowedOrigins to allow any origin, header, and method, facilitating cross-origin requests.Facilitates development and integration with front-end applications hosted on different domains, improving flexibility and developer experience.

```
services.AddCors(options =>
{
    options.AddPolicy("AllowedOrigins",
        builder =>
        {
            builder.AllowAnyOrigin()
                   .AllowAnyMethod()
                   .AllowAnyHeader();
        });
});
```

- Dependency Injection:Registered IUserService and IUserRepository with their respective implementations (UserService and UserRepository) using scoped lifetime.Follows the Dependency Injection principle, promoting loose coupling and enhancing testability and maintainability.

```
services.AddScoped<IUserService, UserService>();
services.AddScoped<IUserRepository, UserRepository>();
```

UserController.cs

1.User Retrieval by Query:

Description: Added an endpoint to fetch a list of users based on query parameters with pagination and search functionality.(Search is based on one parameter- email)

```
GET /users
```

Sample Request:

```
GET /users?search=john@example.com&page=1&pageSize=10
```

Sample Response:

```
[
    {
        "id": "123e4567-e89b-12d3-a456-426614174000",
        "firstName": "John",
        "lastName": "Doe",
        "email": "john.doe@example.com"
    },
    {
        "id": "789e4567-e89b-12d3-a456-426614174000",
        "firstName": "Jane",
        "lastName": "Doe",
        "email": "jane.doe@example.com"
    }
]
```

Motivation: Allows fetching of users based on email and pagination, improving the efficiency of retrieving user lists and enhancing user management capabilities. The search is based on email because email is a unique identifier; two people may have the same first name and last name, but their emails will always be unique. This ensures accurate search results. Partial search is also supported, making it easier to find users even if only part of their email is known

2. User Retrieval by ID:

Description: Added an endpoint to fetch user details by their ID.

```
GET /users/{id:guid}
```

Sample Request:

```
GET /users/123e4567-e89b-12d3-a456-426614174000
```

Sample Response:

```
{
    "id": "123e4567-e89b-12d3-a456-426614174000",
    "firstName": "John",
    "lastName": "Doe",
    "email": "john.doe@example.com"
}
```

Motivation: Provides a way to retrieve specific user details efficiently, improving data retrieval processes and enhancing the overall user experience.

3. User Creation:

Description: Added an endpoint to create new users with validation.

Sample Request:

```
{
    "firstName": "Jane",
    "lastName": "Doe",
    "email": "jane.doe@example.com"
}
```

Sample Response:

```
{
    "id": "789e4567-e89b-12d3-a456-426614174000",
    "firstName": "Jane",
    "lastName": "Doe",
    "email": "jane.doe@example.com"
}
```

Motivation: Allows the creation of new user records while ensuring input data is validated, promoting data integrity and reducing errors.

4.User Update:

Description: Added an endpoint to update existing user details with validation.

Sample Request:

```
{
    "firstName": "Jane",
    "lastName": "Smith",
    "email": "jane.smith@example.com"
}
```

Sample Response:

```
{
    "id": "789e4567-e89b-12d3-a456-426614174000",
    "firstName": "Jane",
    "lastName": "Smith",
    "email": "jane.smith@example.com"
}
```

Motivation: Enables updating of user records with validation to ensure data integrity, making sure only valid data is processed.

5.User Deletion:

Description: Added an endpoint to delete users by their ID.

Sample Request:

```
DELETE /users/789e4567-e89b-12d3-a456-426614174000
```

Sample Response:

```
{
    "message": "User deleted successfully."
}
```

Motivation: Provides a means to remove user records from the system, improving the management and cleanup of user data.

IUserService and UserService

1. Refactor to Use Service Layer:
    - Description: The controller now uses `IUserService` to handle business logic instead of directly interacting with the repository.
    - Motivation: Promotes a clean separation of concerns by delegating business logic to the service layer, enhancing maintainability and testability.
2. Async/Await Implementation:
    - Description: All methods have been updated to use `async/await` for asynchronous operations.
    - Motivation: Improves the application's responsiveness and scalability, particularly under heavy load by allowing non-blocking operations.
3. Model State Validation:
    - Description: Added model state validation in `Create` and `Update` methods to ensure incoming data is valid.
    - Motivation: Ensures that only valid data is processed, reducing the risk of errors and maintaining data integrity.
4. Improved Routing:
    - Description: Updated routes to follow RESTful conventions (e.g., `/users/{id:guid}`).
    - Motivation: Makes the API more intuitive and easier to use for clients by adhering to widely accepted RESTful practices.
5. Comprehensive Error Handling:
    - Description: Added exception handling to provide meaningful error messages.
    - Motivation: Enhances user experience and system reliability by improving overall application robustness.

Using a service layer, as facilitated by the `IUserService` interface, promotes a clean separation of concerns by isolating business logic from the controller. This enhances maintainability and testability by allowing business rules to be managed independently of presentation logic. The `IUserService` interface ensures loose coupling and flexibility, enabling easy modification and extension of business logic without affecting other parts of the application. This design pattern also supports better scalability and reusability of code components.`UserService` encapsulate the logic for retrieving, creating, and updating users, thereby centralizing business logic and promoting a cleaner, more organized codebase. This setup improves testability and allows for easier maintenance and scalability.

UserService.cs

Constructor Dependency Injection:

Purpose: To inject the IUserRepository dependency into the UserService class. This follows the Dependency Injection principle, which promotes loose coupling and makes the code more testable and maintainable.

GetAll Method:

- Purpose: To retrieve all users based on the provided query object.
- Functionality: Calls the repository method to get all users, maps the retrieved entities to DTOs, and returns the list of DTOs.
- Reasoning: Using DTOs ensures that only the necessary data is exposed to the client, improving security and performance.

GetById Method:

- Purpose: To retrieve a user by their unique ID.
- Functionality: Calls the repository method to get the user by ID, maps the entity to a DTO, and returns the DTO. Returns null if the user is not found.
- Reasoning: Mapping to DTOs ensures data encapsulation and decouples the internal data structure from the API contract.

CreateAsync Method:

- Purpose: To create a new user.
- Functionality: Validates the uniqueness of the email, maps the DTO to an entity, saves the entity to the repository, and returns the created user's DTO.
- Reasoning: Ensures that each user has a unique email to prevent duplicate entries and potential conflicts. Using DTOs and entities maintains a clear separation between different layers of the application.

UpdateAsync Method:

- Purpose: To update an existing user's details.
- Functionality: Validates the input, updates the user in the repository, and returns the updated user's DTO. Handles and logs validation exceptions, key not found exceptions, and other unexpected exceptions.
- Reasoning: Proper exception handling ensures that meaningful error messages are provided, and the system remains robust and resilient to errors.

IUserRepository and UserRepository

Using the `IUserRepository` interface and `UserRepository` implementation, the application ensures a clean separation between business logic and data access layers. This approach encapsulates data operations, promoting loose coupling and flexibility, which enhances maintainability and testability. The `IUserRepository` interface defines the contract for user-related data operations, ensuring that the `UserRepository` class implements the necessary methods to interact with the database in a structured and organized manner.

Refactoring to LINQ (EF Core) from ADO.NET

The original implementation of the `UserRepository` used ADO.NET for data access. This approach involved manually handling SQL connections, commands, and data readers. While ADO.NET provides fine-grained control over SQL operations, it comes with certain drawbacks that make it less suitable for modern development practices.

The refactored implementation uses LINQ with Entity Framework Core (EF Core), a modern Object-Relational Mapping (ORM) framework that abstracts away the direct interaction with the database. This transition brings several key benefits:

1. Security- ADO.NET makes the code vulnerable to SQL injection attacks due to concatenated SQL strings. Whereas EF Core uses parameterized queries by default, which mitigates SQL injection risks and enhances overall security.

2. Performance- ADO.NET: requires manual optimization and management of SQL queries and connections, which can be error-prone and less efficient. On the other hand,EF Core: Optimizes queries automatically and manages connections efficiently. It also supports lazy loading, eager loading, and explicit loading, allowing for better performance management.

3. Pagination and Search Functionality- ADO.NET is in need of manual implementation of pagination and search functionality. EF Core supports these features natively, making it easier to implement and manage.

UserRepository.cs

Constructor

- Purpose: Initializes the `UserRepository` class.
- OOP Concept: Encapsulation

GetAllAsync Method

- Purpose: Retrieves all users based on the query object with pagination and search functionality.
- Functionality: Executes a SQL query, paginates results, and maps data to `User` objects.
- Reasoning: Enhances performance and user experience for large datasets by efficiently retrieving user lists.
- OOP Concept: Encapsulation

GetByIdAsync Method

- Purpose: Retrieves a user by their unique ID.
- Functionality: Executes a SQL query and maps data to a `User` object.
- Reasoning: Necessary for detailed data retrieval to provide specific user information.
- OOP Concept: Encapsulation

CreateAsync Method

- Purpose: Creates a new user.
- Functionality: Executes a SQL query to insert a user into the database.
- Reasoning: Ensures new users can be added, maintaining data integrity and supporting user creation functionality.
- OOP Concept: Encapsulation

EmailExistsAsync Method

- Purpose: Checks if an email exists in the database, with an optional parameter to exclude a specific user ID.
- Functionality: Executes a SQL query to count users with the given email. If an optional user ID is provided, it excludes that user ID from the count.
- Reasoning: Validates email uniqueness to prevent duplicates and ensures no conflicts with other users by excluding the specific user ID during updates.
- OOP Concept: Encapsulation, Function Overloading

Data Transfer Objects (DTOs)

The `CreateUserRequestDto` is used for creating new user records. It includes several validation attributes to ensure data integrity. This DTO ensures that all necessary fields are provided and correctly formatted before a new user record is created in the system.Here is the list of Fields and Validations:

- first_name:
  - Validation:`[Required(ErrorMessage = "First name is required.")]`

    `[StringLength(80, ErrorMessage = "First name cannot be longer than 50 characters.")]`
  - Purpose: Ensures that the first name is provided and does not exceed the specified length, maintaining consistency and preventing data entry errors.
- last_name:
  - Validation:
  - `[Required(ErrorMessage = "Last name is required.")]`

    `[StringLength(80, ErrorMessage = "Last name cannot be longer than 50 characters.")]`
  - Purpose: Ensures that the last name is provided and does not exceed the specified length, ensuring proper data entry.
- email:
  - Validation:

    `[Required(ErrorMessage = "Email is required.")]`

    `[EmailAddress(ErrorMessage = "Invalid email address.")]`
  - Purpose: Ensures that a valid email address is provided, which is crucial for user identification and communication.
- date_created:
  - Validation: `[Required]`
  - Purpose: Automatically set to the current date and time, ensuring that the creation timestamp is recorded.

The `UpdateUserRequestDto` is used for updating existing user records. Unlike the `CreateUserRequestDto`, the fields in this DTO are optional, allowing for partial updates. It includes validation attributes to ensure that any provided updates are valid.

- Fields and Validations:
  - first_name:
    - Validation: `[StringLength(80, ErrorMessage = "First name cannot be longer than 50 characters.")]`
    - Purpose: Ensures that the first name, if provided, does not exceed the specified length, maintaining data integrity during updates.
  - last_name:
    - Validation: `[StringLength(80, ErrorMessage = "Last name cannot be longer than 50 characters.")]`
    - Purpose: Ensures that the last name, if provided, does not exceed the specified length, maintaining consistency.
  - email:
    - Validation: `[EmailAddress(ErrorMessage = "Invalid email address.")]`
    - Purpose: Ensures that the email address, if provided, is valid, preventing data entry errors.

The `UserDto` is used for transferring user data between the server and client. It includes all the essential fields that represent a user, formatted consistently to ensure reliable data exchange.

By implementing these DTOs and their associated validations, the application ensures that data integrity is maintained across different operations, from creating new users to updating existing ones and transferring user data between the server and client. These DTOs work seamlessly with mappers, which convert entities to DTOs and vice versa, ensuring that the data structure used in business logic is appropriately transformed for client consumption. Additionally, they integrate with validation helpers, promoting consistency and reducing the risk of errors across the application.

Frontend

Example.tsx

The `Example` component provides an interface to display, search, add, edit, and delete users. It manages user data with pagination and search functionalities, allowing efficient data handling and user management.

Methodology

- React Component Lifecycle:
  - componentDidMount: This method is called when the component is first rendered. It sets the document title and loads the initial user data by calling `this.loadUsers()`.
  - componentDidUpdate: This method is called when the component updates, such as when the pagination or search term changes. It reloads the user data if the `currentPage`, `pageSize`, or `searchTerm` state changes.
- State Management: The component manages its state to control pagination, search, and modal visibility. State variables include `users`, `currentPage`, `pageSize`, `searchTerm`, `hasMoreUsers`, `showModal`, and `editingUser`.
  - These variables are updated using methods like `handlePrevious`, `handleNext`, `handleSearchChange`, `handleEditUser`, and others.
- API Integration: The component integrates with API functions (`fetchUsers`, `fetchUserById`, `updateUser`, `deleteUser`) to handle data operations.
  - loadUsers: Fetches users from the API based on the current page, page size, and search term.
  - handleDeleteUser: Deletes a user by their ID and updates the state to remove the user from the list.
  - handleEditUser: Fetches a user by their ID for editing and displays the edit modal.
  - handleSaveUser: Saves the updated user information and reloads the user list.
- Debouncing Search: The component implements debouncing for the search input to minimize API calls.
  - handleSearchChange: Updates the search term state and uses a debounce timeout to delay the API call, reducing the number of API requests.

Exam.tsx

The `Exam` component provides a user form to input and validate user details (first name, last name, and email). It allows users to submit their details to be added to the system, ensuring data integrity and providing immediate feedback.

Methodology

- State Management:
  - The component manages its state to control form inputs, validation errors, and display of success or error messages.
  - State variables include `firstName`, `lastName`, `email`, `errors`, `showSuccessCard`, `showErrorCard`, and `errorMessage`.
  - State updates are handled using methods like `handleChange`, `validate`, and others to ensure the component reacts appropriately to user inputs and actions.
- Form Validation:
  - validate: Implements validation logic to check the format and presence of required fields.
    - Ensures that `firstName` and `lastName` are not empty and do not contain numbers or special characters.
    - Ensures that `email` is not empty and follows a valid email format.
  - The validation results are stored in the `errors` state variable, and validation messages are displayed next to the respective input fields.
- API Integration:
  - handleSubmit: Uses the `fetch` API to send form data to the backend and handle responses appropriately.
    - Submits the form data (first name, last name, and email) to the `addUser` function, which sends a POST request to the backend.
    - On successful submission, it displays a success message.
    - If an error occurs during submission, it displays an error message.
- Error Handling:
  - Handles potential errors during form submission and provides feedback to the user.
  - Sets the `showErrorCard` state to true and updates the `errorMessage` state with the error details.

Button.tsx

The `Button` component is a reusable component that encapsulates the behavior and styling of a button. It is designed to be flexible and customizable, allowing it to be used in various parts of the application.

Methodology

- Props Management:
  - onClick: A callback function to handle button click events.
  - type: The type of the button, which can be 'button' or 'submit'. Defaults to 'button'.
  - className: Additional CSS classes to apply custom styles to the button. Defaults to an empty string.
  - children: The content inside the button, allowing for flexible use cases (e.g., text, icons).
  - disabled: A boolean indicating whether the button is disabled. Defaults to false.
- Default Props:
  - The component defines default props for `type`, `className`, and `disabled` to ensure default behavior and styling if no props are provided.

Api.tsx

The API integration module defines several functions to interact with the backend API for managing users. It includes functions for fetching, creating, updating, and deleting users. Each function handles the necessary HTTP requests and responses, ensuring that data operations are performed efficiently and correctly.

- Isolates API interaction logic from the component logic, promoting a clean separation of concerns.
- Reusability: Provides reusable functions for various CRUD operations, reducing code duplication.
- Error Handling: Ensures proper error handling for all API interactions, improving the robustness of the application.