Question - 1
.NET Core: News Feed Cache

The company is launching the service, that can manage the news items. The service should be a web API layer using .NET core 3.0. You already have prepared infrastructure and need to implement a web API controller - "*NewsFeedController*". Also, you need to think about performance. The service end-point for getting all available news potentially can take much time in case of large data. Therefore, you need to implement caching for this end-point using in-memory caching mechanism from .NET core.

The following APIs is already implemented:

1. *Creating news* - POST request to endpoint *api/newsfeed* adds the news item to the database. The HTTP response code is 200.
2. *Getting all news* - GET request to endpoint *api/newsfeed* returns the entire list of news. The HTTP response code is 200.
3. *Getting news item by id* - GET request to endpoint *api/newsfeed/{id}* returns the details of the news item for the *{id}*. If there is no news item for the *{id}*, status code 404 is returned. On success, status code 200 is returned.
4. *Getting all new filtered by authorName property* - GET request to endpoint *api/newsfeed?AuthorNames={AuthorName}* returns the entire list of news for *AuthorName*. The HTTP response code is 200.
5. *Deleting a news item by id* - DELETE request to endpoint *api/newsfeed/{id}* delete the corresponding news item. If there is no news item for *{id},* then return status code 404. On success, return status code 204.

Change the APIs end-points of the project such that:
For "Getting all news" end-point you need to think about performance. The first query to GET, hits the database. For the second query to GET all news, you need to get the response faster using .NET core **in-memory cache** mechanism. To perceive the difference between first and second requests, service takes care of adding delay of 2 seconds to imitate a long query to the database.
Important: Any operation that changes the news list should delete the in memory cache. Tests take care of testing this.

Definition of News model:

1. id - ID of the news item. [INTEGER]
2. title - Title of the news item. [STRING]
3. body - Content of the news item. [STRING]
4. authorName - Author name of the news item. [STRING]
5. allowComments - Flag that shows if the news item's comments are available. [BOOLEAN]
6. dateCreated - Date when the news item was created in UTC (GMT + 0). [EPOCH INTEGER]

▼ Example requests and responses

Request 1:
`GET` request to `api/newsfeed`

The response code will be 200 with list of news feed item's details with time delay 2 seconds:

```
[
{
    id: 1,
    title: "news title 1",
    authorName: "Rick D.",
    body: "Advantage old had otherwise sincerity dependent additions.",
    allowComments: true,
    dateCreated: 1573843210
}
]
```

Request 2:
`GET` request to `api/newsfeed`

The response code will be 200 with list of news feed item's details without time delay 2 seconds because used memory cache mechanism:

```
[
{
    id: 1,
    title: "news title 1",
    authorName: "Rick D.",
    body: "Advantage old had otherwise sincerity dependent additions.",
    allowComments: true,
    dateCreated: 1573843210
}
]
```

Request 3:
`DELETE` request to `api/newsfeed/1`

The response code will be 204 and should clean memory cache of the news feed.

Question - 2
.NET: Car Cache

A company is launching a service that can manage cars. The service should be a web API layer using .NET. You already have a prepared infrastructure and need to implement a Web API Controller *CarsController*. Also, you need to think about performance. The service endpoint for getting all available cars potentially can take a long time in case of large data. Therefore, you need to implement caching for this endpoint using the in-memory caching mechanism from .NET.

The following API calls are already implemented:

1. *Creating cars:* a POST request to the endpoint *api/cars* adds a car to the database. The HTTP response code is 200.
2. *Getting all cars:* a GET request to the endpoint *api/cars* returns the entire list of cars. The HTTP response code is 200.
3. *Getting cars item by id:* a GET request to the endpoint *api/cars/{id}* should return the details of the car for the {id}. If there is no car for the {id}, status code 404 is returned. On success, status code 200 is returned.
4. *Getting all cars filtered by the years property:* GET request to the endpoint *api/cars?years={year1}&years={year2}* should return the entire list of cars for *year1* and *year2*. The HTTP response code is 200.
5. *Deleting a car by id:* a DELETE request to the endpoint *api/cars/{id}* should delete the corresponding car. If there is no car for {id}, status code 404 is returned. On success, status code 200 is returned.

Change the API endpoints of the project in the following way:

- For the "Getting all cars" endpoint, you need to think about performance. The first query to GET hits the database. For the second query to GET all cars, you need to get the response faster using the .NET Core in-memory cache mechanism. To perceive the difference between the first and second requests, the service takes care of adding a delay of 2 seconds to imitate a long query to the database.
- Important: Any operation that changes the cars list should delete the in-memory cache. Tests take care of testing this.

Definition of Car model:

1. *id* - The ID of the car. [INTEGER]
2. *make* - The make of the car. [STRING]
3. *model* - The model of the car. [STRING]
4. *price* - The price of the car. [INTEGER]
5. *year* - The car's production year. [INTEGER]

▼ Object's metadata with examples.

Request 1:
`GET` request to `api/cars`

The response code will be 200 with a list of the car's details with a time delay of 2 seconds:

```
[
{
    id: 5,
    make: "Audi",
    model: "A6",
    price: "76000",
    year: 2018
}
]
```

Request 2:
`GET` request to `api/cars`

The response code will be 200 with a list of the car's details without a time delay of 2 seconds because the memory cache mechanism was used:

```
[
{
    id: 5,
    make: "Audi",
    model: "A6",
    price: "76000",
    year: 2018
}
]
```

Request 3:
`DELETE` request to `api/cars/1`

The response code will be 204, and this should clean the memory cache of the news feed.

Question - 3
.NET: File processing Web-API

A company is launching a new service that works with XML files, that contains information about employees, and procures analytics from that. A file service needs to be created, and as part of this challenge, you are required to come up with a service to maintain these XML files.

As step 1, create a service that supports *get* statistics based on XML files. The Test XML file contains employee data such as *First Name, Last Name, Email, Rate*, etc.. Another API endpoint should add to the file user collection and return processed files with added users.

The XML has the following structure:

1. *first_name* - First Name of the employee. [STRING]
2. *last_name* - Last Name of the employee. [STRING]
3. *email* - Email of the Employee. [STRING]
4. *rate* - Rate per hour. [INTEGER]
5. *id* - Unique ID of the employee. [INTEGER]

▼ Example XML structure

<UserCollection>

```
    <User>
      <id>1</id>
      <first_name>Zed</first_name>
      <last_name>Grassi</last_name>
      <email>zgrassi0@youku.com</email>
      <rate>20</rate>
    </User>
</UserCollection>
```

▼ Example JSON statistic model

MODEL:

```
public class StatisticalModel

{

    public float AverageRate { get; set; }

    public float MaxRate { get; set; }

    public float MinRate { get; set; }

}
```

EXAMPLE RESPONSE:

```
{
"AverageRate" : 30
"MaxRate" : 40
"MinRate" : 20
}
```

▼ APIs

The following APIs need to be implemented:

1. *Getting analytics from file* - POST request should be created to add analyze file. The API endpoint would be *api/analyze*. The request body contains the XML file (byte array content) in body. HTTP response should be Status200OK.
2. *Adding employee collection to file* - POST request should be created to add employees to the file. The API endpoint would be *api/adduser*. The request body contains "content" as a string(file content in base64) and employee collection. HTTP response should be Status200OK.

Also, *TestMiddlewareController* contains 2 implemented GET methods. with next routes: */api/testmiddleware/token and /api/testmiddleware/notoken.* Need to create middleware that helps to:

1. Return status code 403 forbidden when call */api/testmiddleware/token* without token.
2. Return status code 200 when call */api/testmiddleware/notoken* without token.
3. Return status code 200 when call */api/testmiddleware/token?token=12345678* without token.

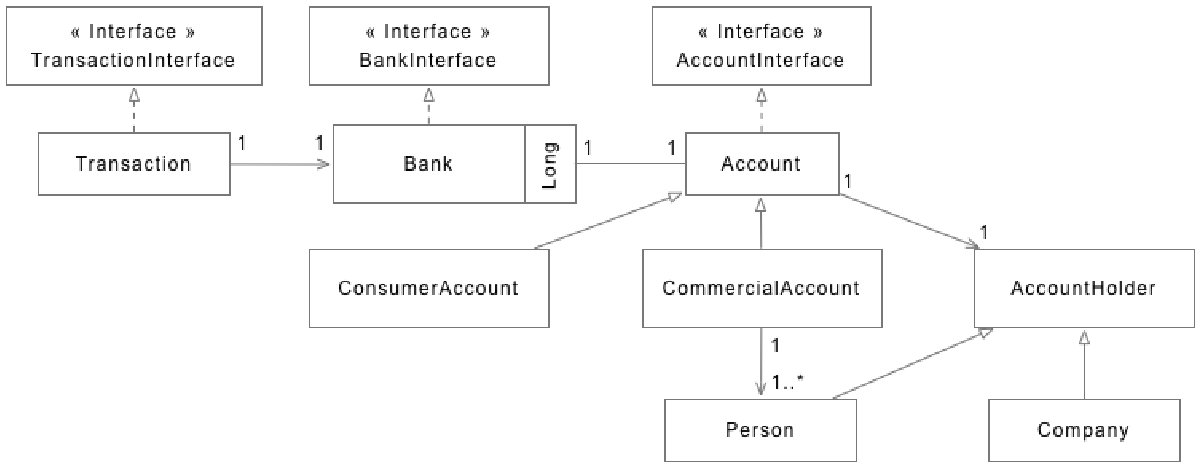
PS. TestMiddlewareController is read-only and can't be changed.

Question - 1
.Net: Banking System

SCORE: **75 points**

.NET C# Classes Abstract Class Constructors Medium

Implement a simple banking system shown in the following domain model.



```
Left-click to open the fully expressed UML diagram in a new tab.
```

System functionality

- Each Transaction object is associated with a Bank and an Account.
- A Bank has a Long to Account map of client account numbers to Account objects.
- Each Account has an AccountHolder, an account number, a pin number, and a balance.
- The AccountHolder class is abstract and is implemented by the Person and Company classes.
- The Account class is abstract and is implemented by the CommercialAccount and ConsumerAccount classes.
- A ConsumerAccount has a single Person as the owner
- A CommercialAccount has a Company as its owner and maintains a list of Person objects that are authorized users on the Account.

Bank provides the following operations using class `Bank.cs`.

1. Withdraw funds from the account
2. Deposit funds to the account
3. Check the balance in the account
4. Get account details
5. Authenticate the user using a PIN
6. Open a consumer account
7. Open a commercial account

There are skeleton methods in `Bank.cs` for these operations. The implementation should follow these rules.

- withdraw: Return true if withdraw amount > available balance, otherwise return false.
- authentication: Return true if PIN matches, otherwise return false.
- new account: A sequential account ID should be assigned to each new account.

A transaction manages these operations using `Transaction.cs`.

1. Withdraw funds from the account.
2. Deposit Funds to the account.
3. Check the balance in the account.

There are skeleton methods in `Transaction.cs` for these operations. The implementation should follow these rules.

- an attempt to perform a transaction in an account with an invalid PIN must throw an `Exception`.

The given .Net project contains a skeleton implementation of the simple banking system shown in the UML diagram and it contains the necessary files and method declarations. Complete the implementation so that the project passes all the unit tests.

Question - 2
.NET Core: Movie Service Localization

SCORE: 75 points

Medium | Back-End Development | Localization | Middleware | .NET

A company is launching an API service that can manage movies. The service should be a web API layer using .NET Core 3.0. You already have a prepared infrastructure, and Web API Controller *MoviesController* is already implemented. You need to think about the localization of the movie's data in response. The language to be used is retrieved from the request header's "Accept-Language" value.

The following API calls are already implemented:
1. *Creating movies*: a POST request to the endpoint *api/movies* adds the movie to the database. The HTTP response code is 200.
2. *Getting movie by id*: a GET request to the endpoint *api/movies/{id}* returns the details of the movie having {id} as a unique identifier. If there is no movie with {id}, response code 404 is returned. On success, the response code is 200.

Change the *SetLanguageMiddleware.cs* and *CustomStringLocalizer.cs* files in the following way:
- Implement the localization mechanism for the movie's "category" property by using the language set in the request header's "Accept-Language" value.
- The service should support 3 languages: "en"- English, "ru"- Russian, and "it"- Italian. Localized (translated) strings are given in the file *CustomStringLocalizer.cs*. Please use the .NET Core IStringLocalizer as a localizer to find the localized string.
- If nothing is set in the request header's "Accept-Language" value, then use the "en" language by default.
- Example: If the request header's "Accept-Language" value is "it" (Italian), then the response should return the movie's "category" value with an Italian translation.

Definition of Movie model:
1. *id* - The ID of the movie. [INTEGER]
2. *title* - The title of the movie. [STRING]
3. *category* - The category to which the movie belongs. [STRING]
4. *releaseDate* - The date when the movie was released in UTC (GMT + 0). [EPOCH INTEGER]

▼ Example requests and responses with headers

Request 1:
GET request to `api/movies/1` without an "Accept-Language" header.

The response code will be 200 with the movie's details. The category value is returned in English because there is no "Accept-Language" header and English is the default language.

```
{
    "id": 1,
    "title": "Friends",
    "category": "Comedy",
    "releaseDate": 1573843210
}
```

Request 2:
GET request to `api/movies/1` with the "Accept-Language" header set to "it".

The response code will be 200 with the movie's details. The category value is returned in the Italian language because the "Accept-Language" header is set to "it".

```
{
    "id": 1,
    "title": "Friends",
    "category": "Commedia",
    "releaseDate": 1573843210
}
```

Question - 3

SCORE: 75 points

.NET Core: Reporting Service With Translations

Back-End Development | Localization | Middleware | Medium | .NET

A company is launching an API service that can manage reports. The service should be a web API layer using .NET. You already have a prepared infrastructure and need to implement a Web API Controller *ReportsController*. Also, you need to think about the localization for the reports' data in responses. The language to be used is retrieved from the request header's "Accept-Language" value.

The following API calls are already implemented:

1. *Creating reports:* a POST request to the endpoint *api/reports* adds a report to the database. The HTTP response code is 200.
2. *Getting all reports:* a GET request to the endpoint *api/reports* returns the entire list of reports. The HTTP response code is 200.

Change the *SetLanguageMiddleware.cs* and *CustomStringLocalizer.cs* files in the following way:

- Implement the localization mechanism for the report's data property *rows* by using the language set in the request header's "Accept-Language" value.
- The service should support 3 languages: "en"- English, "ru"- Russian, and "it"- Italian. Localized (translated) strings are given in the file *CustomStringLocalizer.cs*. Please use the .NET Core IStringLocalizer as a localizer to find the localized string.
- If nothing is set in the request header's "Accept-Language" value, then use the "en" language by default.
- Example: If the request header's "Accept-Language" value is "it" (Italian), then the response should return the report's data property *rows* with Italian translations.

Definition of Report model:

1. *id* - The ID of the report. [INTEGER]
2. *name* - The name of the report. [STRING]
3. *category* - The category of the report. [STRING]
4. *rows* - The report's data. [STRING[]]

▼ Example requests and responses with headers

Request 1:
GET request to `api/reports/1` without an "Accept-Language" header.

The response code will be 200 with the report's details. The property *rows* is filled using the English language because there is no "Accept-Language" header and English is the default language.

```
{
    id: 1,
    name: "Annual report",
    category: "Finance",
    rows: ["Header 1", "Header 2", "Header 3"]
}
```

Request 2:
GET request to `api/reports/1` with the "Accept-Language" header set to "ru".

The response code will be 200 with the report's details. The property *rows* is filled using the Russian language because the "Accept-Language" header set to "ru".

```
{
    id: 1,
    name: "Annual report",
    category: "Finance",
    rows: ["Заголовок 1", "Заголовок 2", "Заголовок 3"]
}
```

Question - 4

SCORE: 75 points

.NET Core: Room Service Authorization

Medium | Back-End Development | Middleware | .NET

A company is launching a room service for managing rooms. The service should be a web API layer using .NET. You already have a prepared infrastructure and need to implement the Web API Controller *RoomsController*. Use the Middleware mechanism from .Net to protect the API service from the third-party requests.

The following API calls are implemented:

1. Creating rooms: a POST request to the endpoint *api/rooms* adds the room to the database. The HTTP response code is 200.
2. Getting all rooms: a GET request to the endpoint *api/rooms* returns the entire list of rooms. The HTTP response code is 200.
3. Getting room by *id*: a GET request to the endpoint *api/rooms/{id}* returns the details of the room for *{id}*. If there is no room with *{id}*, status code 404 is returned. On success, status code 200 is returned.
4. Getting all rooms filtered by the *Floor* property: a GET request to the endpoint *api/rooms?Floors={floor1}&Floors={floor2}* returns the entire list of rooms for *floor1* and *floor2*. The HTTP response code is 200.

Complete the *Middleware* of the project in the following way:

- Implement *Middleware* that checks if the request contains the header *passwordKey* with the value 'passwordKey123456789'. If the request contains this header, allow the request. If not, return HTTP response code 403 Forbidden.

Definition of the *Room* model:

- *id* - The ID of the room. [INTEGER]
- *category* - The category of the room. [STRING]
- *number* - The number of the room. [INTEGER]
- *floor* - The floor of the room. [INTEGER]
- *isAvailable* - The flag that shows if the room is available. [BOOLEAN]
- *addedDate* - The date when the room was added in UTC (GMT + 0). [EPOCH INTEGER]

▼ Example requests and responses with headers

Request 1:

`POST` request to `api/rooms`

The request contains the header *passwordKey* with the value 'passwordKey123456789':

```
{
    id: 1,
    category: "Luxe",
    number: 122,
    floor: 3,
    isAvailable: true,
    addedDate: 1573843210
}
```

The response code will be 200 because the document was created.

Request 2:

`GET` request to `api/rooms/1`

The request doesn't contain the header *passwordKey* with the value 'passwordKey123456789'. Therefore, the response code will be 403.

Request 3:

`GET` request to `api/rooms/1`

The request contains the header *passwordKey* with the value 'passwordKey123456789'. The response code will be 200 with the room's details:

```
{
    id: 1,
    category: "Luxe",
    number: 122,
    floor: 3,
    isAvailable: true,
    addedDate: 1573843210
}
```

Question - 5

SCORE: 75 points

.NET Core: Library Request Counter

Medium | Back-End Development | Middleware | .NET

A company is launching a library service for managing books. The service should be a web API layer using .NET. You already have a prepared infrastructure and need to use the middleware mechanism from .Net Core to keep the track of the total count of requests to this API service.

The following API calls are implemented:

1. *Creating books:* a POST request to the endpoint *api/books* adds the book to the database. The HTTP response code is 200.
2. *Getting all books:* a GET request to the endpoint *api/books* returns the entire list of books. The HTTP response code is 200.
3. *Getting book by id:* a GET request to the endpoint *api/books/{id}* returns the details of the book for {id}. If there is no book with {id}, status code 404 is returned. On success, status code 200 is returned.
4. *Getting all books filtered by the authorName property:* a GET request to the endpoint *api/books?AuthorNames={AuthorName1}&AuthorNames={AuthorName2}* returns the entire list of books for *AuthorName1* and *AuthorName2*. The HTTP response code is 200.

Complete the middleware of the project in the following way:

- **The middleware** needs to track the number of calls to this back-end service. For this, it maintains the counter and keeps incrementing the counter value by 1 on every request. For each request, return the current value of the counter in the response header with the name "requestCounter". The middleware should set the counter to 0 on project/API configuration.

Note: Integration tests take care of testing the middleware. Each integration test sets up the API service again before its execution (as a pre-execution step), so for each integration test, the counter value starts from 0. For example, if you make 3 calls to the API service per 1 integration test, then the response for the last request in this test contains the header "requestCounter" with the value 3.

Definition of the Book model:

1. *id* - The ID of the book. [INTEGER]
2. *title* - The title of the book. [STRING]
3. *body* - The content of the book. [STRING]
4. *authorName* - The name of the book's author. [STRING]
5. *publishedDate* - The date when the book was published in UTC (GMT + 0). [EPOCH INTEGER]

▼ Example requests and responses with headers

Request 1:

`POST` request to `api/books`

Request body:

```
{
    id: 1,
    title: "title 1",
    body: "Advantage old had otherwise sincerity dependent additions.",
    authorName: "Rick D.",
    publishedDate: 1573843210
}
```

The response code will be 200 because the document was created. It also contains the header "requestCounter" with the value 1 because the service endpoint is called only once.

Request 2:

`GET` request to `api/books/1`

The response code will be 200 with the book's details:

```
{
    id: 1,
    title: "title 1",
    body: "Advantage old had otherwise sincerity dependent additions.",
    authorName: "Rick D.",
    publishedDate: 1573843210
}
```

It also contains the header "requestCounter" with the value 2 because the service endpoint is called twice (POST request 1 for creating, and GET request 2 for getting a book).

Question - 1
.NET: Shopping Cart Checkout Validation

Develop two endpoints for a one-page checkout system. The service will be responsible for validating customer information and order data before submitting the order. The primary focus is on ensuring the correctness of the provided information according to specified rules.

Below are the definitions and detailed requirements list. The application will be graded on whether it accurately validates customer information and order data according to the specified rules and successfully handles the submission process.

Customer Information Requirements

- Name: string - The name of the customer.
- Phone: string - The phone number of the customer.
- Email: string - The email address of the customer.
- Address: string - The address of the customer.

Order Information Requirements

- ProductID: int - The ID of the product being ordered.
- Quantity: int - The quantity of the product being ordered.

Validation Rules

1. Customer Information Validation:
    - Validate that all required fields (Name, Phone, Email, Address) are provided.
    - Validate the format and content of each field according to the specified rules.
        - Customer name must consist of more than 2 words.
        - Phone number should contain only digits and be between 3 and 12 digits long.
        - Address must consist of more than 2 words.
2. Order Information Validation:
    - Validate that all required fields (Product ID, Quantity) are provided.
    - Validate the format and content of each field according to the specified rules.
    - Quantity must be greater than 1.
3. Customer and Product Existence Validation:
    - Ensure that the customer is unique based on the email address.
    - Ensure that the customer's name and address are more than 2 words.
    - Ensure that the product ID exists in the product table.
    - The product ID must be a number.

Endpoint 1: Checkout Validation
POST request to /api/checkout/validate:

- Accepts customer and order information in the request body.
- Validates the provided customer information and order data according to the specified rules.
- On success:
    - Returns a success response with HTTP status code 200.
- On failures:
    - Returns an error response with HTTP status code 400 and details of validation errors.

Endpoint 2: Submit Order

POST request to /api/checkout/addOrder:

- Accepts customer and order information in the request body.
- Validates the provided customer information and order data according to the specified rules.
- If the validation is successful:
  - Adds the order to the system.
  - Returns a JSON response containing customer information, order information, and total price.
- If the validation fails:
  - Returns an error response with HTTP status code 400 and details of validation errors.

▼ Example Requests and Responses

Example Request JSON Object

```
{
    "Customer":
    {
        "Name": "Alex Smith Doe",
        "Phone": "1234567890",
        "Email": "alex.doe@example.com",
        "Address": "123 Main Street, City"
    },
    "Products":
    [
        {
            "ProductID": 1,
            "Quantity": 2
        },
        {
            "ProductID": 2,
            "Quantity": 2
        }
    ]
}
```

Example Response JSON Object (Submit Order Success)

```
{
    "Status": 1,
    "Data":
    {
        "Order":
        {
            "OrderID": 1,
            "Total": 50.00,
            "Customer":
            {
                "CustomerID": 1,
                "Name": "Alex Smith Doe",
                "Phone": "1234567890",
                "Email": "alex.doe@example.com",
                "Address": "123 Main Street, City"
            },
            "Products":
            [
                {
                    "ProductID": 1,
                    "Name": "Samsung A24",
                    "Price": 550
                },
                {
                    "ProductID": 2,
                    "Name": "IPhone X",
                    "Price": 120
                }
            ]
        }
    }
}
```

Example Response JSON Object (Validation Failure)

```
{
    "Status": 0,
    "Message": "Order validation errors",
    "Errors":
    [
        "The customer name must contain more than two words.",
        "The customer address must contain more than two words.",
        "The customer phone number format is incorrect.",
        "Product ID does not exist."
    ]
}
```

Question - 2
.NET: Shopping Cart Maintenance

Develop an endpoint for cart maintenance. The service will handle various operations related to managing shopping carts, including adding items to the cart, updating quantities, removing items, and clearing the cart.

Below are the definitions and detailed requirements list. The application will be graded on whether it accurately performs cart maintenance operations according to the specified rules.

Cart Information
A cart will have the following attributes:

- CartID: int - Unique identifier for the cart.
- CartItems: Array of items in the cart.
  - ItemID: int - Unique identifier for the item.
  - Name: string - Name of the item.
  - Quantity: int - Quantity of the item in the cart.
  - Price: float - Price of the item.
- TotalAmount: float - Total amount of all items in the cart.

Operations

- Add Item to Cart:
  - Accepts item information in the request body.
  - Validates the item information.
  - Adds the item to the cart or increments its quantity if it already exists.
  - Updates the total amount in the cart.
  - Returns the updated cart information.
- Update Item Quantity:
  - Accepts item ID and updated quantity in the request body.
  - Validates the updated quantity.
  - Updates the quantity of the specified item in the cart.
  - Updates the total amount in the cart.
  - Returns the updated cart information.
- Remove Item from Cart:
  - Accepts item ID in the request body.
  - Removes the specified item from the cart.
  - Updates the total amount in the cart.
  - Returns the updated cart information.

- Clear Cart:
    - Removes all items from the cart.
    - Resets the total amount in the cart to zero.
    - Returns an empty cart.

Validation Rules

1. Quantity Validation:
    - Validate that the quantity of items is a positive integer.

Endpoint

The service should expose an endpoint for cart maintenance operations. The endpoint should accept different HTTP methods to perform add, update, remove, and clear operations.

The endpoint should follow the following format:

- POST request to /api/cart/add/{id}: Add an item to the cart.
- PUT request to /api/cart/update/{id}/{itemId}/{quantity}: Update the quantity of an item in the cart.
- DELETE request to /api/cart/remove/{id}/{itemId}: Remove an item from the cart.
- DELETE request to /api/cart/clear/{id}: Clear the cart.

▼ Example Requests and Responses

Add Item to Cart

Example request:

POST request to /api/cart/add/1 with the following JSON payload:

```
{
    "Name": "Product A",
    "Quantity": 2,
    "Price": 10.99
}
```

Example response:

```
{
    "CartID": 1,
    "CartItems":
    [
        {
            "ItemID": 1,
            "Name": "Product A",
            "Quantity": 2,
            "Price": 10.99
        }
    ],
    "TotalAmount": 21.98
}
```

Update Item Quantity

Example request:

PUT request to /api/cart/update/1/1/3

Example response:

```
{
    "CartID": 1,
    "CartItems":
    [
        {
            "ItemID": 1,
            "Name": "Product A",
            "Quantity": 3,
            "Price": 10.99
        }
    ],
    "TotalAmount": 32.97
}
```

Remove Item from Cart
Example request:
DELETE request to /api/cart/remove/1/1

Example response:

```
{
    "CartID": 1,
    "CartItems": [],
    "TotalAmount": 0
}
```

Clear Cart
Example request:
DELETE request to /api/cart/clear/1

Example response:

```
{
    "CartID": 1,
    "CartItems": [],
    "TotalAmount": 0
}
```

Question - 3
.NET: User Registration API Service

As part of a company developing security solutions, you are tasked with demonstrating the user registration actions of the API for a new project. The job is to complete Users API endpoints and related UserService classes. Repositories have already been created and use EF Core to perform CRUD operations. The user password will be stored in a database as a hashed string. There is a HashHelper class that will use the HashPassword method to create hashes from passwords. Usernames must be unique.

There are predefined exception classes like AppException and NotFoundException with custom exception middleware. You can throw out these exceptions for convenient cases. The AppException will be returned with 400, and a NotFoundException will be returned with the 404 HTTP status code.

There are different types of objects for different API endpoints. As parameters, the user register endpoint takes a RegisterRequest object, the user update endpoint takes an UpdateRequest object, and the user login endpoint takes a LoginRequest object.

Each User object is a JSON object with the following keys:
- `Id` : the id of the user [int]
- `FirstName` : the first name of the user [string]
- `LastName` : the last name of the user [string]
- `Username` : the login name of the user [string]
- `PasswordHash` : the hashed password of the user [string]

Each RegisterRequest object is a JSON object with the following keys:
- `FirstName` : the first name of the user [string]
- `LastName` : the last name of the user [string]
- `Username` : the login name of the user [string]
- `Password` : the password from the user [string]

Each UpdateRequest object is a JSON object with the following keys:

- `FirstName` : the first name of the user [string]
- `LastName` : the last name of the user [string]
- `Username` : the login name of the user [string]
- `OldPassword` : the current password of the user [string]
- `NewPassword` : the current password of the user [string]

Each LoginRequest object is a JSON object with the following keys:

- `Username` : the login name of the user [string]
- `Password` : the password from the user [string]

Example of a User model JSON object:

```
{
  "Id": 1,
  "FirstName": "John",
  "LastName": "Doe",
  "Username": "JohnDoe1",
  "PasswordHash":"5f4dcc3b5aa765d61d8327deb882cf99"
}
```

Example of a UpdateRequest model JSON object:

```
{
  "FirstName": "Josh",
  "LastName": "Doe",
  "Username": "JohnDoe1",
  "OldPassword":"password",
  "NewPassword":"password23"
}
```

The REST service must expose the `/api/users/` endpoint, which allows the management of the collection of product records.

`POST` request to `/api/users/register` :

- registers a new user to the collection with hashed password
- the response code is 200 and the response data is a User object
- returns 400 if the password is empty or the username is already taken

`POST` request to `/api/users/login` :

- checks the user is already registered
- the response code is 200
- returns 400 if the username or password is empty and returns 404 if the username or password is wrong.

`GET` request to `/api/users` :

- returns all users in the database
- the response code is 200

`GET` request to `/api/users/<id>` :

- returns the user record with the given id
- If the matching record exists, the response code is 200, and the response body is the matching record.
- If there is no record in the collection with the given id, the response code is 404.

`PUT` request to `/api/users/<id>` :

- updates a user data record with the given Id of the user
- The response code is 200.
- It returns 404 if there is no user with the given id, returns 400 if the old password is wrong, and returns 400 if the username update request is for a new name that is already taken by someone else.

`DELETE` request to `/api/users/<id>/` :

- deletes the user record with the given id from the collection
    - If a matching record exists, the response code is 200.
    - If the record with that id does not exist, the response code is 404.

Question - 4
.NET: Product-Order API Service

You are developing an E-Commerce store API. Complete the Products and Orders API endpoints using related service classes. Additionally, implement the *CreateOrder* method in the *OrderService* from the Services folder. Repositories have been created and use Entity Framework Core to perform CRUD operations.

Creating orders takes a *CreateOrderModel* object that has a list of *OrderProductModel*, which has only *ProductId* and *Quantity* properties. You must find related products, calculate the total price, and save it in the order object.

Model Definitions

Each *product* object is a JSON object with the following keys:
- *Id*: The ID of the *product* [int]
- *Name*: The name of the *product* [string]
- *Price*: The price of the *product* [decimal]

Each *order* object is a JSON object with the following keys:
- *Id*: The ID of the *order* [int]
- *TotalPrice*: The sum of the *product prices in the order* [decimal]
- *OrderDate*: The date of the *order* [DateTime]
- *OrderProducts*: The list of the products in the *order* [OrderProduct]

Each *orderProduct* object is a JSON object with the following keys:
- *Id*: The ID of the *orderProduct* object [int]
- *OrderId*: The ID of the *order* [int]
- *ProductId*: The ID of the *product* [int]
- *Quantity*: The quantity of the *product* [int]
- *Price*: The price of the single *product* [decimal]

Example JSON Objects

Example of a *Product* model JSON object:

```
{
  "Id": 1,
  "Name": "Sample product",
  "Price": 5.49
}
```

Example of an *Order* model JSON object:

```
{
  "id": 1,
  "totalPrice": 35,
  "orderDate": "2023-04-03T18:50:13.3227462+03:00",
  "orderProducts": [
    {"id": 2, "orderId": 1, "productId": 48, "quantity": 3, "price": 10.22},
    {"id": 1, "orderId": 1, "productId": 13, "quantity": 1, "price": 5.45}
  ]
}
```

Example of a *CreateOrderModel* JSON object:

```
{"OrderProducts":[{"ProductId":1,"Quantity":1},{"ProductId":19,"Quantity":4}]}
```

API Endpoints

The Products REST service must expose the */api/products/* endpoint, which allows for managing the collection of product records in the following way:
- POST request to */api/products/*: Creates a new product data record. The response code is 200, and the response data is a product object with ID.
- GET request to */api/products/*: The response code is 200, and the body is an array of all the products in the collection.

- GET request to */api/products/<id>*: The response code is 200, and the body is a product record with the given ID. If there is no record in the collection with the given ID, the response code is 404 (NotFound).
- PUT request to */api/products/<id>*: Updates a product data record with the given ID of the product and the response code is 204 (NoContent). If there is no record in the collection with the given ID, the response code is 404. If the ID in the parameter and the object do not match, the response code is 400 (BadRequest).
- DELETE request to */api/products/<id>/*: Deletes the product record with the given ID from the collection. If a matching record existed, the response code is 204. Otherwise, the response code is 404.

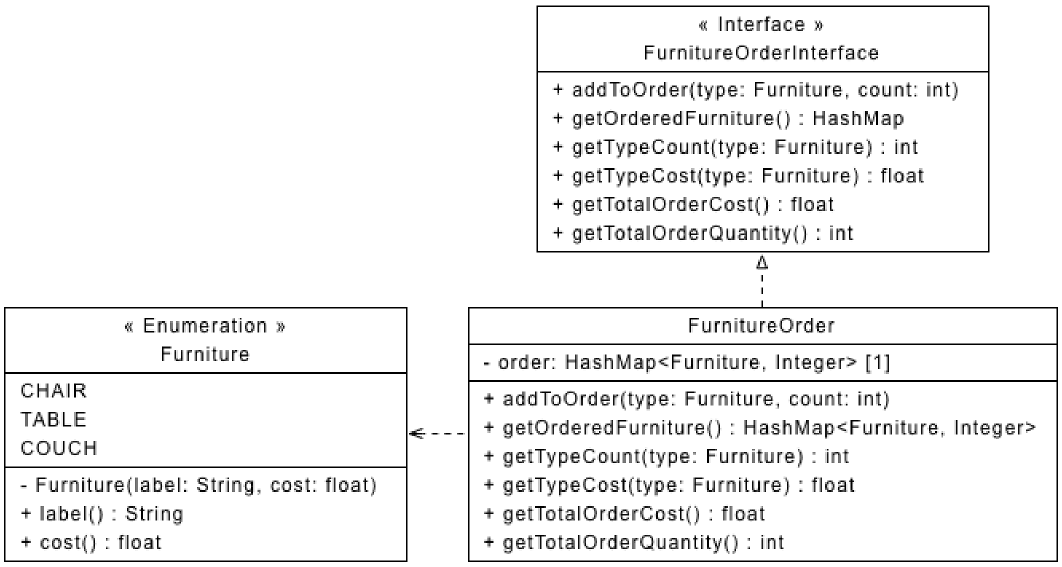
The Orders REST service must expose the */api/orders/* endpoint, which allows for managing the collection of order records in the following way:

- POST request to */api/orders/*: Takes *CreateOrderModel* from the request body. Creates a new order data record. The response code is 200, and the response data is an order object with ID.
- GET request to */api/orders/*: The response code is 200, and the body is an array of all the orders in the collection.
- GET request to */api/orders/<id>*: The response code is 200, and the body is an *order* record with the given ID. If there is no record in the collection with the given ID, the response code is 404 (NotFound).
- PUT request to */api/orders/<id>*: Updates an order data record with the given ID of the *order*, and the response code is 204 (NoContent). If there is no record in the collection with the given ID, the response code is 404. If the ID in the parameter and the object do not match, the response code is 400 (BadRequest).
- DELETE request to */api/orders/<id>/*: Deletes the *order* record with the given ID from the collection. If a matching record existed, the response code is 204. Otherwise, the response code is 404.

Question - 5

.Net: Furniture Factory

Consider this UML class diagram.


« Interface »
FurnitureOrderInterface
+ addToOrder(type: Furniture, count: int)
+ getOrderedFurniture() : HashMap
+ getTypeCount(type: Furniture) : int
+ getTypeCost(type: Furniture) : float
+ getTotalOrderCost() : float
+ getTotalOrderQuantity() : int
« Enumeration »
Furniture
CHAIR
TABLE
COUCH
- Furniture(label: String, cost: float)
+ label() : String
+ cost() : float
FurnitureOrder
- order: HashMap<Furniture, Integer> [1]
+ addToOrder(type: Furniture, count: int)
+ getOrderedFurniture() : HashMap<Furniture, Integer>
+ getTypeCount(type: Furniture) : int
+ getTypeCost(type: Furniture) : float
+ getTotalOrderCost() : float
+ getTotalOrderQuantity() : int

A .Net Core 3.1 project containing the following is provided.

- The implementation and documentation for `Furniture.cs` and `FurnitureOrderInterface.cs`.
- An incomplete implementation of `FurnitureOrder.cs`.
- Expected functionality of each method in the `FurnitureOrderInterface.cs` is described in the comments of corresponding methods.

Complete the implementation of `FurnitureOrder.cs` so that the project passes all the unit tests.