.NET Microservices – Azure DevOps and AKS

Section 6: Orders Microservice - Notes

Orders Microservice:

Code Explanation - Entities

Order Class:

Purpose: The Order class represents an order in an eCommerce system, including details like order ID, user ID, order date, total bill, and a list of items included in the order.

```
using MongoDB.Bson.Serialization.Attributes;


namespace eCommerce.OrdersMicroservice.DataAccessLayer.Entities
{
    // Represents an order in the system
    public class Order
    {
        // The unique identifier for the document in MongoDB
        [BsonId]
        [BsonRepresentation(MongoDB.Bson.BsonType.String)]
        public Guid _id { get; set; }

        // Unique identifier for the order
        [BsonRepresentation(MongoDB.Bson.BsonType.String)]
        public Guid OrderID { get; set; }

        // Unique identifier for the user who placed the order
        [BsonRepresentation(MongoDB.Bson.BsonType.String)]
        public Guid UserID { get; set; }

        // The date when the order was placed
        [BsonRepresentation(MongoDB.Bson.BsonType.String)]
```

```
        public DateTime OrderDate { get; set; }


        // The total amount billed for the order
        [BsonRepresentation(MongoDB.Bson.BsonType.Double)]
        public decimal TotalBill { get; set; }


        // List of items included in the order
        public List<OrderItem> OrderItems { get; set; } = new List<OrderItem>();
    }
}
```

Explanation:

1. **using MongoDB.Bson.Serialization.Attributes;**
    - This directive allows the use of MongoDB-specific attributes that help in mapping C# classes to MongoDB documents.
2. **namespace eCommerce.OrdersMicroservice.DataAccessLayer.Entities**
    - Defines the namespace for organizing code and preventing name conflicts.
3. **public class Order**
    - Declares a public class named Order. The public access modifier means that this class can be accessed from other assemblies.
4. **[BsonId]**
    - Specifies that the following property is the primary key of the MongoDB document. MongoDB uses this to uniquely identify each document.
5. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Indicates that the Guid property should be stored as a string in MongoDB. This helps with the representation of Guid in MongoDB, which doesn't have a direct Guid type.
6. **public Guid _id { get; set; }**
    - Declares a property named _id of type Guid. This serves as the primary key for the MongoDB document.
7. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Again, specifies that the following property should be stored as a string in MongoDB.
8. **public Guid OrderID { get; set; }**
    - Declares a property for storing the unique order identifier. This is the ID that identifies an order within the system.

9. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Specifies that the Guid should be stored as a string in MongoDB.
10. **public Guid UserID { get; set; }**
    - Declares a property for storing the unique user identifier who placed the order.
11. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Specifies that the DateTime should be stored as a string in MongoDB.
12. **public DateTime OrderDate { get; set; }**
    - Declares a property for storing the date when the order was placed.
13. **[BsonRepresentation(MongoDB.Bson.BsonType.Double)]**
    - Specifies that the decimal should be stored as a double in MongoDB. MongoDB does not have a direct decimal type, so double is used as an approximation.
14. **public decimal TotalBill { get; set; }**
    - Declares a property for storing the total bill amount for the order.
15. **public List<OrderItem> OrderItems { get; set; } = new List<OrderItem>();**
    - Declares a property for storing a list of OrderItem objects associated with the order. Initializes it with an empty list to ensure it's never null.

OrderItem Class:

Purpose: The OrderItem class represents an individual item within an order, including details like the product ID, unit price, quantity, and total price.

```
using MongoDB.Bson.Serialization.Attributes;


namespace eCommerce.OrdersMicroservice.DataAccessLayer.Entities
{
   // Represents an item in an order
   public class OrderItem
   {
      // The unique identifier for the document in MongoDB
      [BsonId]
      [BsonRepresentation(MongoDB.Bson.BsonType.String)]
      public Guid _id { get; set; }
```

```
        // Unique identifier for the product
        [BsonRepresentation(MongoDB.Bson.BsonType.String)]
        public Guid ProductID { get; set; }

        // The price of one unit of the product
        [BsonRepresentation(MongoDB.Bson.BsonType.Double)]
        public decimal UnitPrice { get; set; }

        // The quantity of the product ordered
        [BsonRepresentation(MongoDB.Bson.BsonType.Int32)]
        public int Quantity { get; set; }

        // The total price for the ordered quantity of the product
        [BsonRepresentation(MongoDB.Bson.BsonType.Double)]
        public decimal TotalPrice { get; set; }
    }
}
```

Explanation:

1. **using MongoDB.Bson.Serialization.Attributes;**
    - Imports the necessary attributes for mapping C# properties to MongoDB document fields.
2. **namespace eCommerce.OrdersMicroservice.DataAccessLayer.Entities**
    - Defines the namespace for the OrderItem class.
3. **public class OrderItem**
    - Declares a public class named OrderItem. This class represents an item within an order.
4. **[BsonId]**
    - Marks the following property as the primary key for the MongoDB document.
5. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Specifies that the Guid property should be represented as a string in MongoDB.

6. **public Guid _id { get; set; }**
    - Declares a property named _id to serve as the primary key for the OrderItem document.
7. **[BsonRepresentation(MongoDB.Bson.BsonType.String)]**
    - Specifies that the Guid should be represented as a string in MongoDB.
8. **public Guid ProductID { get; set; }**
    - Declares a property for storing the unique identifier of the product.
9. **[BsonRepresentation(MongoDB.Bson.BsonType.Double)]**
    - Specifies that the decimal value should be stored as a double in MongoDB.
10. **public decimal UnitPrice { get; set; }**
    - Declares a property for storing the price per unit of the product.
11. **[BsonRepresentation(MongoDB.Bson.BsonType.Int32)]**
    - Specifies that the int value should be stored as an integer in MongoDB.
12. **public int Quantity { get; set; }**
    - Declares a property for storing the quantity of the product ordered.
13. **[BsonRepresentation(MongoDB.Bson.BsonType.Double)]**
    - Specifies that the decimal value should be stored as a double in MongoDB.
14. **public decimal TotalPrice { get; set; }**
    - Declares a property for storing the total price for the ordered quantity of the product.

Code Explanation - Repository Contracts

IOrdersRepository Interface

Purpose: The IOrdersRepository interface defines the contract for interacting with the Order collection in the data access layer. It provides methods for CRUD operations and querying the Order collection.

```
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

using MongoDB.Driver;


namespace eCommerce.OrdersMicroservice.DataAccessLayer.RepositoryContracts

{
```

```
// Defines the contract for operations related to Orders in the data layer
public interface IOrdersRepository
{
    /// <summary>
    /// Retrieves all Orders asynchronously
    /// </summary>
    /// <returns>Returns all orders from the orders collection</returns>
    Task<IEnumerable<Order>> GetOrders();

    /// <summary>
    /// Retrieves all Orders based on the specified condition asynchronously
    /// </summary>
    /// <param name="filter">The condition to filter orders</param>
    /// <returns>Returning a collection of matching orders</returns>
    Task<IEnumerable<Order?>> GetOrdersByCondition(FilterDefinition<Order> filter);

    /// <summary>
    /// Retrieves a single order based on the specified condition asynchronously
    /// </summary>
    /// <param name="filter">The condition to filter Orders</param>
    /// <returns>Returning matching order</returns>
    Task<Order?> GetOrderByCondition(FilterDefinition<Order> filter);

    /// <summary>
    /// Adds a new Order into the Orders collection asynchronously
    /// </summary>
    /// <param name="order">The order to be added</param>
    /// <returns>Returnes the added Order object or null if unsuccessful</returns>
    Task<Order?> AddOrder(Order order);

    /// <summary>
```

```
    /// Updates an existing order asynchronously
    /// </summary>
    /// <param name="order">The order to be added</param>
    /// <returns>Returns the updated order object; or null if not found</returns>
    Task<Order?> UpdateOrder(Order order);

    /// <summary>
    /// Deletes the order asynchronously
    /// </summary>
    /// <param name="orderID">The Order ID based on which we need to delete the order</param>
    /// <returns>Returns true if the deletion is successful, false otherwise</returns>
    Task<bool> DeleteOrder(Guid orderID);
  }
}
```

Explanation:

1. **using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;**
    - Imports the Order class from the Entities namespace to be used in this interface.
2. **using MongoDB.Driver;**
    - Imports MongoDB's driver library for using MongoDB-specific types and operations.
3. **namespace eCommerce.OrdersMicroservice.DataAccessLayer.RepositoryContracts**
    - Defines the namespace for organizing repository contract interfaces.
4. **public interface IOrdersRepository**
    - Declares a public interface named IOrdersRepository. Interfaces define contracts for classes that implement them.
5. **/// <summary> ... /// <returns>**
    - XML comments used to document the purpose and behavior of each method in the interface. These comments are used by documentation tools and IDEs to provide descriptions and generate API documentation.
6. **Task<IEnumerable<Order>> GetOrders();**
    - Defines a method to asynchronously retrieve all orders. Returns a Task that, when completed, will provide an IEnumerable<Order> representing all orders in the collection.

7. **Task<IEnumerable<Order?>> GetOrdersByCondition(FilterDefinition<Order> filter);**
    - Defines a method to asynchronously retrieve orders that match a specific filter condition. Uses FilterDefinition<Order> from MongoDB to specify the filter.
8. **Task<Order?> GetOrderByCondition(FilterDefinition<Order> filter);**
    - Defines a method to asynchronously retrieve a single order that matches a filter condition. Returns a single Order object or null if no matching order is found.
9. **Task<Order?> AddOrder(Order order);**
    - Defines a method to asynchronously add a new order to the collection. Returns the added Order object or null if the addition was unsuccessful.
10. **Task<Order?> UpdateOrder(Order order);**
    - Defines a method to asynchronously update an existing order. Returns the updated Order object or null if the order was not found.
11. **Task<bool> DeleteOrder(Guid orderID);**
    - Defines a method to asynchronously delete an order based on its OrderID. Returns true if the deletion was successful, otherwise false.

Code Explanation - Repository

OrdersRepository Class

```
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;
using eCommerce.OrdersMicroservice.DataAccessLayer.RepositoryContracts;
using MongoDB.Driver;

namespace eCommerce.OrdersMicroservice.DataAccessLayer.Repositories
{
    public class OrdersRepository : IOrdersRepository
    {
        private readonly IMongoCollection<Order> _orders;
        private readonly string collectionName = "orders";

        public OrdersRepository(IMongoDatabase mongoDatabase)
        {
```

```
        _orders = mongoDatabase.GetCollection<Order>(collectionName);
    }

    public async Task<Order?> AddOrder(Order order)
    {
        order.OrderID = Guid.NewGuid(); // Assign a new unique OrderID
        order._id = order.OrderID; // Set _id to the same value as OrderID

        foreach (OrderItem orderItem in order.OrderItems)
        {
            orderItem._id = Guid.NewGuid(); // Assign a new unique ID to each order item
        }

        await _orders.InsertOneAsync(order); // Insert the order into the collection
        return order; // Return the inserted order
    }

    public async Task<bool> DeleteOrder(Guid orderID)
    {
        FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, orderID); // Create filter to find order

        Order? existingOrder = (await _orders.FindAsync(filter)).FirstOrDefault(); // Find the order that matches the filter

        if (existingOrder == null)
        {
            return false; // Return false if the order was not found
        }

        DeleteResult deleteResult = await _orders.DeleteOneAsync(filter); // Delete the order that matches the filter
```

```
        return deleteResult.DeletedCount > 0; // Return true if the deletion was successful, otherwise false
    }

    public async Task<Order?> GetOrderByCondition(FilterDefinition<Order> filter)
    {
        return (await _orders.FindAsync(filter)).FirstOrDefault(); // Find and return a single order that matches the filter
    }

    public async Task<IEnumerable<Order>> GetOrders()
    {
        return (await _orders.FindAsync(Builders<Order>.Filter.Empty)).ToList(); // Find and return all orders
    }

    public async Task<IEnumerable<Order?>> GetOrdersByCondition(FilterDefinition<Order> filter)
    {
        return (await _orders.FindAsync(filter)).ToList(); // Find and return orders that match the filter
    }

    public async Task<Order?> UpdateOrder(Order order)
    {
        FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, order.OrderID); // Create filter to find order

        Order? existingOrder = (await _orders.FindAsync(filter)).FirstOrDefault(); // Find the order that matches the filter

        if (existingOrder == null)
        {
            return null; // Return null if the order was not found
```

```
        }

        order._id = existingOrder._id; // Set the _id of the updated order to the existing order's _id

        ReplaceOneResult replaceOneResult = await _orders.ReplaceOneAsync(filter, order); // Replace the existing order with the updated order

        return order; // Return the updated order
    }
  }
}
```

Explanation:

AddOrder Method

1. **public async Task<Order?> AddOrder(Order order)**
    - Defines an asynchronous method that returns a Task with an Order object, which may be null.
2. **order.OrderID = Guid.NewGuid();**
    - Generates a new unique identifier for the OrderID property of the order using Guid.NewGuid().
3. **order._id = order.OrderID;**
    - Sets the _id field to the same value as OrderID. In MongoDB, _id is the primary key and must be unique.
4. **foreach (OrderItem orderItem in order.OrderItems)**
    - Iterates over each OrderItem in the OrderItems list to assign unique IDs.
5. **orderItem._id = Guid.NewGuid();**
    - Generates a new unique identifier for each OrderItem.
6. **await _orders.InsertOneAsync(order);**
    - Asynchronously inserts the order into the MongoDB collection _orders.
7. **return order;**
    - Returns the inserted order object.

DeleteOrder Method

1. **public async Task<bool> DeleteOrder(Guid orderID)**
    - Defines an asynchronous method that returns a Task with a boolean value indicating success or failure.
2. **FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, orderID);**
    - Creates a filter to find an order based on its OrderID. The Builders<Order>.Filter.Eq method specifies an equality condition.
3. **Order? existingOrder = (await _orders.FindAsync(filter)).FirstOrDefault();**
    - Asynchronously finds the order that matches the filter. FirstOrDefault() returns the first matching order or null if no match is found.
4. **if (existingOrder == null)**
    - Checks if the existingOrder is null, meaning no order was found.
5. **return false;**
    - Returns false if the order was not found.
6. **DeleteResult deleteResult = await _orders.DeleteOneAsync(filter);**
    - Asynchronously deletes the order that matches the filter.
7. **return deleteResult.DeletedCount > 0;**
    - Returns true if the DeletedCount is greater than 0, indicating that at least one document was deleted.

GetOrderByCondition Method

1. **public async Task<Order?> GetOrderByCondition(FilterDefinition<Order> filter)**
    - Defines an asynchronous method that returns a Task with an Order object, which may be null.
2. **return (await _orders.FindAsync(filter)).FirstOrDefault();**
    - Asynchronously finds and returns the first order that matches the filter. FirstOrDefault() returns null if no match is found.

GetOrders Method

1. **public async Task<IEnumerable<Order>> GetOrders()**
    - Defines an asynchronous method that returns a Task with an IEnumerable<Order>.
2. **return (await _orders.FindAsync(Builders<Order>.Filter.Empty)).ToList();**
    - Asynchronously finds and returns all orders. Builders<Order>.Filter.Empty specifies no filter, so all documents are retrieved. ToList() converts the result to a list.

GetOrdersByCondition Method

1. **public async Task<IEnumerable<Order?>> GetOrdersByCondition(FilterDefinition<Order> filter)**
    - Defines an asynchronous method that returns a Task with an IEnumerable<Order?>.
2. **return (await _orders.FindAsync(filter)).ToList();**
    - Asynchronously finds and returns all orders that match the filter. ToList() converts the result to a list.

UpdateOrder Method

1. **public async Task<Order?> UpdateOrder(Order order)**
    - Defines an asynchronous method that returns a Task with an Order object, which may be null.
2. **FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, order.OrderID);**
    - Creates a filter to find an order based on its OrderID.
3. **Order? existingOrder = (await _orders.FindAsync(filter)).FirstOrDefault();**
    - Asynchronously finds the order that matches the filter. FirstOrDefault() returns the first matching order or null if no match is found.
4. **if (existingOrder == null)**
    - Checks if the existingOrder is null, meaning no order was found.
5. **return null;**
    - Returns null if the order was not found.
6. **order._id = existingOrder._id;**
    - Sets the _id field of the order to the _id of the existing order. This ensures that the same document is updated.
7. **ReplaceOneResult replaceOneResult = await _orders.ReplaceOneAsync(filter, order);**

- Asynchronously replaces the existing order with the updated order.

8. **return order;**
    - Returns the updated order object.

Code Explanation - DependencyInjection in DataAccessLayer

DependencyInjection Class

The DependencyInjection class configures and registers services related to data access in the dependency injection container. This setup allows the services to be injected into other components of the application.

```
using eCommerce.OrdersMicroservice.DataAccessLayer.RepositoryContracts;

using eCommerce.OrdersMicroservice.DataAccessLayer.Repositories;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using MongoDB.Driver;


namespace eCommerce.OrdersMicroservice.DataAccessLayer

{

  public static class DependencyInjection

  {

    public static IServiceCollection AddDataAccessLayer(this IServiceCollection services,
IConfiguration configuration)

    {

      //TO DO: Add data access layer services into the IoC container

      string connectionStringTemplate = configuration.GetConnectionString("MongoDB")!;

      string connectionString = connectionStringTemplate

       .Replace("$MONGO_HOST", Environment.GetEnvironmentVariable("MONGODB_HOST"))

       .Replace("$MONGO_PORT", Environment.GetEnvironmentVariable("MONGODB_PORT"));


      services.AddSingleton<IMongoClient>(new MongoClient(connectionString));
```

```
        services.AddScoped<IMongoDatabase>(provider =>
        {
            IMongoClient client = provider.GetRequiredService<IMongoClient>();
            return
client.GetDatabase(Environment.GetEnvironmentVariable("MONGODB_DATABASE"));
        });


        services.AddScoped<IOrdersRepository, OrdersRepository>();


        return services;
    }
  }
}
```

Explanation:

AddDataAccessLayer Method

1. **public static IServiceCollection AddDataAccessLayer(this IServiceCollection services, IConfiguration configuration)**
    - Defines a static extension method for IServiceCollection to register data access layer services. The method accepts IConfiguration for accessing configuration settings.
2. **string connectionStringTemplate = configuration.GetConnectionString("MongoDB")!;**
    - Retrieves the connection string template for MongoDB from the configuration settings. The ! operator is used to assert that the value is not null.
3. **string connectionString = connectionStringTemplate**
    - Initializes a new string connectionString by replacing placeholders with actual environment variable values.
4. **.Replace("$MONGO_HOST", Environment.GetEnvironmentVariable("MONGODB_HOST"))**
    - Replaces the $MONGO_HOST placeholder in the connection string with the value from the environment variable MONGODB_HOST.
5. **.Replace("$MONGO_PORT", Environment.GetEnvironmentVariable("MONGODB_PORT"));**
    - Replaces the $MONGO_PORT placeholder in the connection string with the value from the environment variable MONGODB_PORT.
6. **services.AddSingleton<IMongoClient>(new MongoClient(connectionString));**
    - Registers IMongoClient as a singleton service in the dependency injection container. MongoClient is created using the connection string. Singleton means that only one

instance of IMongoClient will be created and used throughout the application's lifetime.

7. **services.AddScoped<IMongoDatabase>(provider =>**
    - Registers IMongoDatabase as a scoped service. Scoped means that a new instance is created for each request within a scope, such as an HTTP request.
8. **IMongoClient client = provider.GetRequiredService<IMongoClient>();**
    - Retrieves the IMongoClient instance from the service provider. GetRequiredService ensures that the service is available and throws an exception if it is not.
9. **return client.GetDatabase(Environment.GetEnvironmentVariable("MONGODB_DATABASE"));**
    - Uses the IMongoClient instance to get the MongoDB database specified by the environment variable MONGODB_DATABASE.
10. **services.AddScoped<IOrdersRepository, OrdersRepository>();**
    - Registers IOrdersRepository with its implementation OrdersRepository as a scoped service. This means that each request will receive a new instance of OrdersRepository, but the same instance will be used throughout the scope of the request.
11. **return services;**
    - Returns the IServiceCollection for chaining additional service registrations.

Code Explanation - DTO in BusinessLogicLayer

These Data Transfer Object (DTO) classes are used to transfer data between different layers of the application, such as between the API and the business logic layer. Each record class represents a different type of data structure for various operations involving orders and related entities.

OrderAddRequest Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderAddRequest(Guid UserID, DateTime OrderDate, List<OrderItemAddRequest>
OrderItems)

{

  public OrderAddRequest() : this(default, default, default)

  {
```

```
 }
}
```

Explanation:

- **public record OrderAddRequest(Guid UserID, DateTime OrderDate, List<OrderItemAddRequest> OrderItems)**
    - Defines a record type OrderAddRequest that contains data for creating a new order.
    - UserID: The ID of the user placing the order.
    - OrderDate: The date the order was placed.
    - OrderItems: A list of items in the order, each represented by OrderItemAddRequest.
- **public OrderAddRequest() : this(default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

OrderItemAddRequest Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderItemAddRequest(Guid ProductID, decimal UnitPrice, int Quantity)
{
 public OrderItemAddRequest() : this(default, default, default)
 {
 }
}
```

Explanation:

- **public record OrderItemAddRequest(Guid ProductID, decimal UnitPrice, int Quantity)**
    - Defines a record type OrderItemAddRequest for specifying details about an item in an order.
    - ProductID: The ID of the product.
    - UnitPrice: The price per unit of the product.
    - Quantity: The number of units of the product.
- **public OrderItemAddRequest() : this(default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

OrderItemResponse Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderItemResponse(Guid ProductID, decimal UnitPrice, int Quantity, decimal
TotalPrice, string? ProductName, string? Category)
{
  public OrderItemResponse() : this(default, default, default, default, default, default)
  {
  }
}
```

Explanation:

- **public record OrderItemResponse(Guid ProductID, decimal UnitPrice, int Quantity, decimal TotalPrice, string? ProductName, string? Category)**
    - Defines a record type OrderItemResponse for returning item details in responses.
    - ProductID: The ID of the product.
    - UnitPrice: The price per unit of the product.
    - Quantity: The number of units ordered.
    - TotalPrice: The total price for the quantity ordered.
    - ProductName: The name of the product (optional).
    - Category: The category of the product (optional).
- **public OrderItemResponse() : this(default, default, default, default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

OrderItemUpdateRequest Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderItemUpdateRequest(Guid ProductID, decimal UnitPrice, int Quantity)
{
  public OrderItemUpdateRequest() : this(default, default, default)
  {
  }
}
```

Explanation:

- **public record OrderItemUpdateRequest(Guid ProductID, decimal UnitPrice, int Quantity)**
    - Defines a record type OrderItemUpdateRequest for updating details of an existing order item.
    - ProductID: The ID of the product.
    - UnitPrice: The updated price per unit of the product.
    - Quantity: The updated number of units ordered.
- **public OrderItemUpdateRequest() : this(default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

OrderResponse Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderResponse(Guid OrderID, Guid UserID, decimal TotalBill, DateTime OrderDate,
List<OrderItemResponse> OrderItems, string? UserPersonName, string? Email)

{

 public OrderResponse() : this(default, default, default, default, default, default, default)

 {

 }

}
```

Explanation:

- **public record OrderResponse(Guid OrderID, Guid UserID, decimal TotalBill, DateTime OrderDate, List<OrderItemResponse> OrderItems, string? UserPersonName, string? Email)**
    - Defines a record type OrderResponse for returning order details in responses.
    - OrderID: The unique ID of the order.
    - UserID: The ID of the user who placed the order.
    - TotalBill: The total amount for the order.
    - OrderDate: The date the order was placed.
    - OrderItems: A list of items in the order, each represented by OrderItemResponse.
    - UserPersonName: The name of the user who placed the order (optional).
    - Email: The email of the user who placed the order (optional).

- **public OrderResponse() : this(default, default, default, default, default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

OrderUpdateRequest Class

```
namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;


public record OrderUpdateRequest(Guid OrderID, Guid UserID, DateTime OrderDate,
List<OrderItemUpdateRequest> OrderItems)

{

  public OrderUpdateRequest() : this(default, default, default, default)

  {

  }

}
```

Explanation:

- **public record OrderUpdateRequest(Guid OrderID, Guid UserID, DateTime OrderDate, List<OrderItemUpdateRequest> OrderItems)**
    - Defines a record type OrderUpdateRequest for updating an existing order.
    - OrderID: The ID of the order to be updated.
    - UserID: The ID of the user placing the update.
    - OrderDate: The updated date of the order.
    - OrderItems: A list of updated items in the order, each represented by OrderItemUpdateRequest.
- **public OrderUpdateRequest() : this(default, default, default, default)**
    - Defines a parameterless constructor that initializes the record with default values.

ProductDTO Class

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

public record ProductDTO(Guid ProductID, string? ProductName, string? Category, double UnitPrice, int QuantityInStock);

Explanation:

- **public record ProductDTO(Guid ProductID, string? ProductName, string? Category, double UnitPrice, int QuantityInStock)**
    - Defines a record type ProductDTO for transferring product details.
    - ProductID: The ID of the product.
    - ProductName: The name of the product (optional).
    - Category: The category of the product (optional).
    - UnitPrice: The price per unit of the product.
    - QuantityInStock: The quantity of the product in stock.

UserDTO Class

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

public record UserDTO(Guid UserID, string? Email, string? PersonName, string Gender);

Explanation:

- **public record UserDTO(Guid UserID, string? Email, string? PersonName, string Gender)**
    - Defines a record type UserDTO for transferring user details.
    - UserID: The ID of the user.
    - Email: The email address of the user (optional).
    - PersonName: The name of the user (optional).
    - Gender: The gender of the user.

Code Explanation - HttpClient classes in BusinessLogicLayer

These classes, UsersMicroserviceClient and ProductsMicroserviceClient, are responsible for making HTTP requests to other microservices to fetch user and product details, respectively. Each class utilizes HttpClient to send requests and handle responses.

UsersMicroserviceClient Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using System.Net.Http.Json;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.HttpClients;

public class UsersMicroserviceClient
{
 private readonly HttpClient _httpClient;

 public UsersMicroserviceClient(HttpClient httpClient)
 {
  _httpClient = httpClient;
 }


 public async Task<UserDTO?> GetUserByUserID(Guid userID)
 {
  HttpResponseMessage response = await _httpClient.GetAsync($"/api/users/{userID}");

  if (!response.IsSuccessStatusCode)
  {
   if (response.StatusCode == System.Net.HttpStatusCode.NotFound)
   {
    return null;
   }
   else if (response.StatusCode == System.Net.HttpStatusCode.BadRequest)
```

```
    {
      throw new HttpRequestException("Bad request", null,
System.Net.HttpStatusCode.BadRequest);
    }
    else
    {
      throw new HttpRequestException($"Http request failed with status code
{response.StatusCode}");
    }
   }



   UserDTO? user = await response.Content.ReadFromJsonAsync<UserDTO>();


   if (user == null)
   {
    throw new ArgumentException("Invalid User ID");
   }


   return user;
  }
}
```

Explanation:

- **using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;**
    - Imports the DTO namespace to use the UserDTO class.
- **using System.Net.Http.Json;**
    - Imports the System.Net.Http.Json namespace to use extension methods like ReadFromJsonAsync.
- **public class UsersMicroserviceClient**
    - Defines a class UsersMicroserviceClient to handle HTTP requests related to user data.
- **private readonly HttpClient _httpClient;**
    - Declares a private read-only field to store an instance of HttpClient for making HTTP requests.
- **public UsersMicroserviceClient(HttpClient httpClient)**
    - Defines a constructor that initializes _httpClient with an instance provided through dependency injection.
- **public async Task<UserDTO?> GetUserByUserID(Guid userID)**
    - Defines an asynchronous method to get user details by their ID.
- **HttpResponseMessage response = await _httpClient.GetAsync($"/api/users/{userID}");**
    - Sends a GET request to the /api/users/{userID} endpoint to retrieve user data.
- **if (!response.IsSuccessStatusCode)**
    - Checks if the response indicates a failure (status code not in the range 200-299).
- **if (response.StatusCode == System.Net.HttpStatusCode.NotFound)**
    - Returns null if the response status code is 404 Not Found.
- **else if (response.StatusCode == System.Net.HttpStatusCode.BadRequest)**
    - Throws an HttpRequestException with a "Bad request" message if the status code is 400 Bad Request.
- **else**
    - Throws a generic HttpRequestException for other status codes indicating request failure.
- **UserDTO? user = await response.Content.ReadFromJsonAsync<UserDTO>();**
    - Deserializes the response content to an instance of UserDTO.
- **if (user == null)**

- Throws an ArgumentException if deserialization results in null, indicating an invalid user ID.

- **return user;**
  - Returns the deserialized UserDTO object.

ProductsMicroserviceClient Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using System.Net.Http.Json;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.HttpClients;

public class ProductsMicroserviceClient
{
  private readonly HttpClient _httpClient;

  public ProductsMicroserviceClient(HttpClient httpClient)
  {
    _httpClient = httpClient;
  }


  public async Task<ProductDTO?> GetProductByProductID(Guid productID)
  {
    HttpResponseMessage response = await _httpClient.GetAsync($"/api/products/search/product-id/{productID}");

    if (!response.IsSuccessStatusCode)
    {
      if (response.StatusCode == System.Net.HttpStatusCode.NotFound)
      {
        return null;
      }
```

```
    else if (response.StatusCode == System.Net.HttpStatusCode.BadRequest)
    {
      throw new HttpRequestException("Bad request", null,
System.Net.HttpStatusCode.BadRequest);
    }
    else
    {
      throw new HttpRequestException($"Http request failed with status code
{response.StatusCode}");
    }
   }



   ProductDTO? product = await response.Content.ReadFromJsonAsync<ProductDTO>();


   if (product == null)
   {
     throw new ArgumentException("Invalid Product ID");
   }


   return product;
  }
}
```

Explanation:

- **using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;**
    - Imports the DTO namespace to use the ProductDTO class.
- **using System.Net.Http.Json;**
    - Imports the System.Net.Http.Json namespace to use extension methods like ReadFromJsonAsync.
- **public class ProductsMicroserviceClient**
    - Defines a class ProductsMicroserviceClient to handle HTTP requests related to product data.

- **private readonly HttpClient _httpClient;**
    - Declares a private read-only field to store an instance of HttpClient for making HTTP requests.
- **public ProductsMicroserviceClient(HttpClient httpClient)**
    - Defines a constructor that initializes _httpClient with an instance provided through dependency injection.
- **public async Task<ProductDTO?> GetProductByProductID(Guid productID)**
    - Defines an asynchronous method to get product details by their ID.
- **HttpResponseMessage response = await _httpClient.GetAsync($"/api/products/search/product-id/{productID}");**
    - Sends a GET request to the /api/products/search/product-id/{productID} endpoint to retrieve product data.
- **if (!response.IsSuccessStatusCode)**
    - Checks if the response indicates a failure (status code not in the range 200-299).
- **if (response.StatusCode == System.Net.HttpStatusCode.NotFound)**
    - Returns null if the response status code is 404 Not Found.
- **else if (response.StatusCode == System.Net.HttpStatusCode.BadRequest)**
    - Throws an HttpRequestException with a "Bad request" message if the status code is 400 Bad Request.
- **else**
    - Throws a generic HttpRequestException for other status codes indicating request failure.
- **ProductDTO? product = await response.Content.ReadFromJsonAsync<ProductDTO>();**
    - Deserializes the response content to an instance of ProductDTO.
- **if (product == null)**
    - Throws an ArgumentException if deserialization results in null, indicating an invalid product ID.
- **return product;**
    - Returns the deserialized ProductDTO object.

Code Explanation - AutoMapper classes in BusinessLogicLayer

These classes define mapping profiles for converting between various data transfer objects (DTOs) and data access layer entities. The AutoMapper library is used to streamline these conversions, which helps in keeping the business logic clean and focused on its core functionality.

OrderAddRequestToOrderMappingProfile Class

```
using AutoMapper;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;


namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;


public class OrderAddRequestToOrderMappingProfile : Profile

{

  public OrderAddRequestToOrderMappingProfile()

  {

    CreateMap<OrderAddRequest, Order>()

      .ForMember(dest => dest.UserID, opt => opt.MapFrom(src => src.UserID))

      .ForMember(dest => dest.OrderDate, opt => opt.MapFrom(src => src.OrderDate))

      .ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems))

      .ForMember(dest => dest.OrderID, opt => opt.Ignore())

      .ForMember(dest => dest._id, opt => opt.Ignore())

      .ForMember(dest => dest.TotalBill, opt => opt.Ignore());

  }

}
```

Explanation:

- **CreateMap<OrderAddRequest, Order>()**
    - Defines a mapping from OrderAddRequest to Order.
- **.ForMember(dest => dest.OrderID, opt => opt.Ignore())**
    - Specifies that the OrderID property in the Order entity should be ignored (not mapped from OrderAddRequest).
- **.ForMember(dest => dest._id, opt => opt.Ignore())**

- Specifies that the _id property in the Order entity should be ignored.

- **.ForMember(dest => dest.TotalBill, opt => opt.Ignore())**
  - Specifies that the TotalBill property in the Order entity should be ignored.

OrderItemAddRequestToOrderItemMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class OrderItemAddRequestToOrderItemMappingProfile : Profile
{
  public OrderItemAddRequestToOrderItemMappingProfile()
  {
    CreateMap<OrderItemAddRequest, OrderItem>()
      .ForMember(dest => dest.ProductID, opt => opt.MapFrom(src => src.ProductID))
      .ForMember(dest => dest.UnitPrice, opt => opt.MapFrom(src => src.UnitPrice))
      .ForMember(dest => dest.Quantity, opt => opt.MapFrom(src => src.Quantity))
      .ForMember(dest => dest.TotalPrice, opt => opt.Ignore())
      .ForMember(dest => dest._id, opt => opt.Ignore());
  }
}
```

Explanation:

- **CreateMap<OrderItemAddRequest, OrderItem>()**
  - Defines a mapping from OrderItemAddRequest to OrderItem.
- **.ForMember(dest => dest.TotalPrice, opt => opt.Ignore())**
  - Specifies that the TotalPrice property in the OrderItem entity should be ignored.
- **.ForMember(dest => dest._id, opt => opt.Ignore())**
  - Specifies that the _id property in the OrderItem entity should be ignored.

OrderItemToOrderItemResponseMappingProfile Class

```
eusing AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class OrderItemToOrderItemResponseMappingProfile : Profile
{
  public OrderItemToOrderItemResponseMappingProfile()
  {
    CreateMap<OrderItem, OrderItemResponse>()
      .ForMember(dest => dest.ProductID, opt => opt.MapFrom(src => src.ProductID))
      .ForMember(dest => dest.UnitPrice, opt => opt.MapFrom(src => src.UnitPrice))
      .ForMember(dest => dest.Quantity, opt => opt.MapFrom(src => src.Quantity))
      .ForMember(dest => dest.TotalPrice, opt => opt.MapFrom(src => src.TotalPrice));
  }
}
```

Explanation:

- **CreateMap<OrderItem, OrderItemResponse>()**
    - Defines a mapping from OrderItem to OrderItemResponse.

OrderItemUpdateRequestToOrderItemMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class OrderItemUpdateRequestToOrderItemMappingProfile : Profile
{
```

```
  public OrderItemUpdateRequestToOrderItemMappingProfile()
  {
    CreateMap<OrderItemUpdateRequest, OrderItem>()
      .ForMember(dest => dest.ProductID, opt => opt.MapFrom(src => src.ProductID))
      .ForMember(dest => dest.UnitPrice, opt => opt.MapFrom(src => src.UnitPrice))
      .ForMember(dest => dest.Quantity, opt => opt.MapFrom(src => src.Quantity))
      .ForMember(dest => dest.TotalPrice, opt => opt.Ignore())
      .ForMember(dest => dest._id, opt => opt.Ignore());
  }
}
```

Explanation:

- **CreateMap<OrderItemUpdateRequest, OrderItem>()**
    - Defines a mapping from OrderItemUpdateRequest to OrderItem.

OrderToOrderResponseMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class OrderToOrderResponseMappingProfile : Profile
{
  public OrderToOrderResponseMappingProfile()
  {
    CreateMap<Order, OrderResponse>()
      .ForMember(dest => dest.OrderID, opt => opt.MapFrom(src => src.OrderID))
      .ForMember(dest => dest.UserID, opt => opt.MapFrom(src => src.UserID))
      .ForMember(dest => dest.OrderDate, opt => opt.MapFrom(src => src.OrderDate))
      .ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems))
      .ForMember(dest => dest.TotalBill, opt => opt.MapFrom(src => src.TotalBill));
```

```
  }
}
```

Explanation:

- **CreateMap<Order, OrderResponse>()**
    - Defines a mapping from Order to OrderResponse.

OrderUpdateRequestToOrderMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class OrderUpdateRequestToOrderMappingProfile : Profile
{
  public OrderUpdateRequestToOrderMappingProfile()
  {
    CreateMap<OrderUpdateRequest, Order>()
      .ForMember(dest => dest.OrderID, opt => opt.MapFrom(src => src.OrderID))
      .ForMember(dest => dest.UserID, opt => opt.MapFrom(src => src.UserID))
      .ForMember(dest => dest.OrderDate, opt => opt.MapFrom(src => src.OrderDate))
      .ForMember(dest => dest.OrderItems, opt => opt.MapFrom(src => src.OrderItems))
      .ForMember(dest => dest._id, opt => opt.Ignore())
      .ForMember(dest => dest.TotalBill, opt => opt.Ignore());
  }
}
```

Explanation:

- **CreateMap<OrderUpdateRequest, Order>()**
    - Defines a mapping from OrderUpdateRequest to Order.

ProductDTOToOrderItemResponseMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

public class ProductDTOToOrderItemResponseMappingProfile : Profile
{
  public ProductDTOToOrderItemResponseMappingProfile()
  {
    CreateMap<ProductDTO, OrderItemResponse>()
      .ForMember(dest => dest.ProductName, opt => opt.MapFrom(src => src.ProductName))
      .ForMember(dest => dest.Category, opt => opt.MapFrom(src => src.Category));
  }
}
```

Explanation:

- **CreateMap<ProductDTO, OrderItemResponse>()**
    - Defines a mapping from ProductDTO to OrderItemResponse.

UserDTOToOrderResponseMappingProfile Class

```
using AutoMapper;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.Mappers;

public class UserDTOToOrderResponseMappingProfile : Profile
{
  public UserDTOToOrderResponseMappingProfile()
  {
    CreateMap<UserDTO, OrderResponse>()
      .ForMember(dest => dest.UserPersonName, opt => opt.MapFrom(src => src.PersonName))
```

```
    .ForMember(dest => dest.Email, opt => opt.MapFrom(src => src.Email));
  }
}
```

Explanation:

- **CreateMap<UserDTO, OrderResponse>()**
    - Defines a mapping from UserDTO to OrderResponse.

Code Explanation - Validators in BusinessLogicLayer

These validator classes use FluentValidation to enforce rules on various DTOs. They ensure that the input data meets the specified requirements before any further processing is done. This helps in maintaining data integrity and preventing invalid data from being handled by the business logic.

OrderAddRequestValidator Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using FluentValidation;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.Validators;

public class OrderAddRequestValidator : AbstractValidator<OrderAddRequest>
{
  public OrderAddRequestValidator()
  {
    // UserID
    RuleFor(temp => temp.UserID)
      .NotEmpty().WithErrorCode("User ID can't be blank");

    // OrderDate
    RuleFor(temp => temp.OrderDate)
      .NotEmpty().WithErrorCode("Order Date can't be blank");
```

```
    // OrderItems
    RuleFor(temp => temp.OrderItems)
      .NotEmpty().WithErrorCode("Order Items can't be blank");
  }
}
```

Explanation:

- **RuleFor(temp => temp.UserID)**
  - Ensures that the UserID field is not empty.
- **RuleFor(temp => temp.OrderDate)**
  - Ensures that the OrderDate field is not empty.
- **RuleFor(temp => temp.OrderItems)**
  - Ensures that the OrderItems collection is not empty.

OrderItemAddRequestValidator Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using FluentValidation;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.Validators;

public class OrderItemAddRequestValidator : AbstractValidator<OrderItemAddRequest>
{
  public OrderItemAddRequestValidator()
  {
    // ProductID
    RuleFor(temp => temp.ProductID)
      .NotEmpty().WithErrorCode("Product ID can't be blank");

    // UnitPrice
    RuleFor(temp => temp.UnitPrice)
      .NotEmpty().WithErrorCode("Unit Price can't be blank")
      .GreaterThan(0).WithErrorCode("Unit Price can't be less than or equal to zero");
```

```
    // Quantity
    RuleFor(temp => temp.Quantity)
      .NotEmpty().WithErrorCode("Quantity can't be blank")
      .GreaterThan(0).WithErrorCode("Quantity can't be less than or equal to zero");
  }
}
```

Explanation:

- **RuleFor(temp => temp.ProductID)**
  - Ensures that the ProductID field is not empty.
- **RuleFor(temp => temp.UnitPrice)**
  - Ensures that the UnitPrice is not empty and is greater than zero.
- **RuleFor(temp => temp.Quantity)**
  - Ensures that the Quantity is not empty and is greater than zero.

OrderItemUpdateRequestValidator Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using FluentValidation;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.Validators;

public class OrderItemUpdateRequestValidator : AbstractValidator<OrderItemUpdateRequest>
{
  public OrderItemUpdateRequestValidator()
  {
    // ProductID
    RuleFor(temp => temp.ProductID)
      .NotEmpty().WithErrorCode("Product ID can't be blank");

    // UnitPrice
    RuleFor(temp => temp.UnitPrice)
```

```
    .NotEmpty().WithErrorCode("Unit Price can't be blank")
    .GreaterThan(0).WithErrorCode("Unit Price can't be less than or equal to zero");

  // Quantity
  RuleFor(temp => temp.Quantity)
    .NotEmpty().WithErrorCode("Quantity can't be blank")
    .GreaterThan(0).WithErrorCode("Quantity can't be less than or equal to zero");
 }
}
```

Explanation:

- **RuleFor(temp => temp.ProductID)**
  - Ensures that the ProductID field is not empty.
- **RuleFor(temp => temp.UnitPrice)**
  - Ensures that the UnitPrice is not empty and is greater than zero.
- **RuleFor(temp => temp.Quantity)**
  - Ensures that the Quantity is not empty and is greater than zero.

OrderUpdateRequestValidator Class

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using FluentValidation;

namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.Validators;

public class OrderUpdateRequestValidator : AbstractValidator<OrderUpdateRequest>
{
 public OrderUpdateRequestValidator()
 {
  // OrderID
  RuleFor(temp => temp.OrderID)
    .NotEmpty().WithErrorCode("Order ID can't be blank");
```

```
    // UserID
    RuleFor(temp => temp.UserID)
      .NotEmpty().WithErrorCode("User ID can't be blank");

    // OrderDate
    RuleFor(temp => temp.OrderDate)
      .NotEmpty().WithErrorCode("Order Date can't be blank");

    // OrderItems
    RuleFor(temp => temp.OrderItems)
      .NotEmpty().WithErrorCode("Order Items can't be blank");
  }
}
```

Explanation:

- **RuleFor(temp => temp.OrderID)**
  - Ensures that the OrderID field is not empty.
- **RuleFor(temp => temp.UserID)**
  - Ensures that the UserID field is not empty.
- **RuleFor(temp => temp.OrderDate)**
  - Ensures that the OrderDate field is not empty.
- **RuleFor(temp => temp.OrderItems)**
  - Ensures that the OrderItems collection is not empty.

Summary

- **OrderAddRequestValidator**: Validates that UserID, OrderDate, and OrderItems are not empty.
- **OrderItemAddRequestValidator**: Validates that ProductID is not empty, UnitPrice is not empty and greater than zero, and Quantity is not empty and greater than zero.
- **OrderItemUpdateRequestValidator**: Same as OrderItemAddRequestValidator but for update scenarios.
- **OrderUpdateRequestValidator**: Validates that OrderID, UserID, OrderDate, and OrderItems are not empty.

Code Explanation - Service Contracts in BusinessLogicLayer

The IOrdersService interface defines the contract for managing orders in the application. It includes methods for retrieving, adding, updating, and deleting orders. Each method's description helps in understanding its functionality and the expected outcomes. Here's a detailed look at each method in the IOrdersService interface:

IOrdersService Interface

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

using MongoDB.Driver;


namespace eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts;


public interface IOrdersService

{

 /// <summary>

 /// Retrieves the list of orders from the orders repository

 /// </summary>

 /// <returns>Returns list of OrderResponse objects</returns>

 Task<List<OrderResponse?>> GetOrders();


 /// <summary>

 /// Returns list of orders matching with given condition

 /// </summary>

 /// <param name="filter">Expression that represents condition to check</param>

 /// <returns>Returns matching orders as OrderResponse objects</returns>

 Task<List<OrderResponse?>> GetOrdersByCondition(FilterDefinition<Order> filter);


 /// <summary>
```

```
  /// Returns a single order that matches with given condition
  /// </summary>
  /// <param name="filter">Expression that represents the condition to check</param>
  /// <returns>Returns matching order object as OrderResponse; or null if not found</returns>
  Task<OrderResponse?> GetOrderByCondition(FilterDefinition<Order> filter);

  /// <summary>
  /// Add (inserts) order into the collection using orders repository
  /// </summary>
  /// <param name="orderAddRequest">Order to insert</param>
  /// <returns>Returns OrderResponse object that contains order details after inserting; or returns null if insertion is unsuccessful.</returns>
  Task<OrderResponse?> AddOrder(OrderAddRequest orderAddRequest);

  /// <summary>
  /// Updates the existing order based on the OrderID
  /// </summary>
  /// <param name="orderUpdateRequest">Order data to update</param>
  /// <returns>Returns order object after successful updation; otherwise null</returns>
  Task<OrderResponse?> UpdateOrder(OrderUpdateRequest orderUpdateRequest);

  /// <summary>
  /// Deletes an existing order based on given order id
  /// </summary>
  /// <param name="orderID">OrderID to search and delete</param>
  /// <returns>Returns true if the deletion is successful; otherwise false</returns>
  Task<bool> DeleteOrder(Guid orderID);
}
```

Method Details

1. **GetOrders**
    - **Purpose:** Retrieves all orders from the repository.
    - **Returns:** A list of OrderResponse objects representing all orders.
    - **Use Case:** To fetch and display all orders for administrative or analytical purposes.
2. **GetOrdersByCondition**
    - **Purpose:** Retrieves orders that match a specific condition.
    - **Parameters:**
        - filter: A FilterDefinition<Order> representing the condition to filter the orders.
    - **Returns:** A list of OrderResponse objects that match the condition.
    - **Use Case:** To fetch orders based on specific criteria, such as date range, user ID, or status.
3. **GetOrderByCondition**
    - **Purpose:** Retrieves a single order that matches a specific condition.
    - **Parameters:**
        - filter: A FilterDefinition<Order> representing the condition to filter the order.
    - **Returns:** An OrderResponse object if a match is found; otherwise, returns null.
    - **Use Case:** To fetch a specific order based on a unique condition.
4. **AddOrder**
    - **Purpose:** Adds a new order to the repository.
    - **Parameters:**
        - orderAddRequest: An OrderAddRequest object containing the details of the order to be added.
    - **Returns:** An OrderResponse object representing the newly added order, or null if the addition fails.
    - **Use Case:** To insert a new order into the system.
5. **UpdateOrder**
    - **Purpose:** Updates an existing order based on its ID.
    - **Parameters:**
        - orderUpdateRequest: An OrderUpdateRequest object containing the updated order details.

- **Returns:** An OrderResponse object representing the updated order, or null if the update fails.
- **Use Case:** To modify the details of an existing order.

6. **DeleteOrder**
    - **Purpose:** Deletes an order from the repository.
    - **Parameters:**
        - orderID: The ID of the order to be deleted.
    - **Returns:** True if the deletion is successful; otherwise, false.
    - **Use Case:** To remove an order from the system, typically done for order cancellations or deletions.

Code Explanation - OrdersService in BusinessLogicLayer

The OrdersService class implements the IOrdersService interface and provides the business logic for managing orders in the system. This service handles various operations such as adding, updating, deleting, and retrieving orders.

OrdersService:

```
using AutoMapper;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.HttpClients;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts;

using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;

using eCommerce.OrdersMicroservice.DataAccessLayer.RepositoryContracts;

using FluentValidation;

using FluentValidation.Results;

using MongoDB.Driver;


namespace eCommerce.ordersMicroservice.BusinessLogicLayer.Services;


public class OrdersService : IOrdersService

{
```

```
 private readonly IValidator<OrderAddRequest> _orderAddRequestValidator;
 private readonly IValidator<OrderItemAddRequest> _orderItemAddRequestValidator;
 private readonly IValidator<OrderUpdateRequest> _orderUpdateRequestValidator;
 private readonly IValidator<OrderItemUpdateRequest> _orderItemUpdateRequestValidator;
 private readonly IMapper _mapper;
 private IOrdersRepository _ordersRepository;
 private UsersMicroserviceClient _usersMicroserviceClient;
 private ProductsMicroserviceClient _productsMicroserviceClient;

 public OrdersService(IOrdersRepository ordersRepository, IMapper mapper,
IValidator<OrderAddRequest> orderAddRequestValidator, IValidator<OrderItemAddRequest>
orderItemAddRequestValidator, IValidator<OrderUpdateRequest> orderUpdateRequestValidator,
IValidator<OrderItemUpdateRequest> orderItemUpdateRequestValidator, UsersMicroserviceClient
usersMicroserviceClient, ProductsMicroserviceClient productsMicroserviceClient)
 {
  _orderAddRequestValidator = orderAddRequestValidator;
  _orderItemAddRequestValidator = orderItemAddRequestValidator;
  _orderUpdateRequestValidator = orderUpdateRequestValidator;
  _orderItemUpdateRequestValidator = orderItemUpdateRequestValidator;
  _mapper = mapper;
  _ordersRepository = ordersRepository;
  _usersMicroserviceClient = usersMicroserviceClient;
  _productsMicroserviceClient = productsMicroserviceClient;
 }

 public async Task<OrderResponse?> AddOrder(OrderAddRequest orderAddRequest)
 {
  if (orderAddRequest == null)
  {
   throw new ArgumentNullException(nameof(orderAddRequest));
  }
```

```
    ValidationResult orderAddRequestValidationResult = await
_orderAddRequestValidator.ValidateAsync(orderAddRequest);

    if (!orderAddRequestValidationResult.IsValid)

    {

      string errors = string.Join(", ", orderAddRequestValidationResult.Errors.Select(temp =>
temp.ErrorMessage));

      throw new ArgumentException(errors);

    }


    List<ProductDTO?> products = new List<ProductDTO?>();


    foreach (OrderItemAddRequest orderItemAddRequest in orderAddRequest.OrderItems)

    {

      ValidationResult orderItemAddRequestValidationResult = await
_orderItemAddRequestValidator.ValidateAsync(orderItemAddRequest);

      if (!orderItemAddRequestValidationResult.IsValid)

      {

        string errors = string.Join(", ", orderItemAddRequestValidationResult.Errors.Select(temp =>
temp.ErrorMessage));

        throw new ArgumentException(errors);

      }


      ProductDTO? product = await
_productsMicroserviceClient.GetProductByProductID(orderItemAddRequest.ProductID);

      if (product == null)

      {

        throw new ArgumentException("Invalid Product ID");

      }


      products.Add(product);

    }


    UserDTO? user = await _usersMicroserviceClient.GetUserByUserID(orderAddRequest.UserID);
```

```
  if (user == null)
  {
    throw new ArgumentException("Invalid User ID");
  }

  Order orderInput = _mapper.Map<Order>(orderAddRequest);
  foreach (OrderItem orderItem in orderInput.OrderItems)
  {
    orderItem.TotalPrice = orderItem.Quantity * orderItem.UnitPrice;
  }
  orderInput.TotalBill = orderInput.OrderItems.Sum(temp => temp.TotalPrice);

  Order? addedOrder = await _ordersRepository.AddOrder(orderInput);

  if (addedOrder == null)
  {
    return null;
  }

  OrderResponse addedOrderResponse = _mapper.Map<OrderResponse>(addedOrder);

  if (addedOrderResponse != null)
  {
    foreach (OrderItemResponse orderItemResponse in addedOrderResponse.OrderItems)
    {
      ProductDTO? productDTO = products.FirstOrDefault(temp => temp.ProductID ==
orderItemResponse.ProductID);
      if (productDTO != null)
      {
        _mapper.Map(productDTO, orderItemResponse);
      }
```

```
    }

    if (user != null)
    {
      _mapper.Map(user, addedOrderResponse);
    }
   }

   return addedOrderResponse;
  }

  public async Task<OrderResponse?> UpdateOrder(OrderUpdateRequest orderUpdateRequest)
  {
   if (orderUpdateRequest == null)
   {
    throw new ArgumentNullException(nameof(orderUpdateRequest));
   }

   ValidationResult orderUpdateRequestValidationResult = await
_orderUpdateRequestValidator.ValidateAsync(orderUpdateRequest);
   if (!orderUpdateRequestValidationResult.IsValid)
   {
    string errors = string.Join(", ", orderUpdateRequestValidationResult.Errors.Select(temp =>
temp.ErrorMessage));
    throw new ArgumentException(errors);
   }

   List<ProductDTO> products = new List<ProductDTO>();

   foreach (OrderItemUpdateRequest orderItemUpdateRequest in orderUpdateRequest.OrderItems)
   {
```

```
    ValidationResult orderItemUpdateRequestValidationResult = await
_orderItemUpdateRequestValidator.ValidateAsync(orderItemUpdateRequest);

    if (!orderItemUpdateRequestValidationResult.IsValid)

    {

      string errors = string.Join(", ", orderItemUpdateRequestValidationResult.Errors.Select(temp =>
temp.ErrorMessage));

      throw new ArgumentException(errors);

    }


    ProductDTO? product = await
_productsMicroserviceClient.GetProductByProductID(orderItemUpdateRequest.ProductID);

    if (product == null)

    {

      throw new ArgumentException("Invalid Product ID");

    }


    products.Add(product);

  }


  UserDTO? user = await _usersMicroserviceClient.GetUserByUserID(orderUpdateRequest.UserID);

  if (user == null)

  {

    throw new ArgumentException("Invalid User ID");

  }


  Order orderInput = _mapper.Map<Order>(orderUpdateRequest);

  foreach (OrderItem orderItem in orderInput.OrderItems)

  {

    orderItem.TotalPrice = orderItem.Quantity * orderItem.UnitPrice;

  }

  orderInput.TotalBill = orderInput.OrderItems.Sum(temp => temp.TotalPrice);
```

```
    Order? updatedOrder = await _ordersRepository.UpdateOrder(orderInput);

    if (updatedOrder == null)
    {
      return null;
    }

    OrderResponse updatedOrderResponse = _mapper.Map<OrderResponse>(updatedOrder);

    if (updatedOrderResponse != null)
    {
      foreach (OrderItemResponse orderItemResponse in updatedOrderResponse.OrderItems)
      {
        ProductDTO? productDTO = products.FirstOrDefault(temp => temp.ProductID ==
orderItemResponse.ProductID);
        if (productDTO != null)
        {
          _mapper.Map(productDTO, orderItemResponse);
        }
      }

      if (user != null)
      {
        _mapper.Map(user, updatedOrderResponse);
      }
    }

    return updatedOrderResponse;
  }

  public async Task<bool> DeleteOrder(Guid orderID)
```

```
{
  FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, orderID);
  Order? existingOrder = await _ordersRepository.GetOrderByCondition(filter);

  if (existingOrder == null)
  {
    return false;
  }

  bool isDeleted = await _ordersRepository.DeleteOrder(orderID);
  return isDeleted;
}

public async Task<OrderResponse?> GetOrderByCondition(FilterDefinition<Order> filter)
{
  Order? order = await _ordersRepository.GetOrderByCondition(filter);
  if (order == null)
  {
    return null;
  }

  OrderResponse orderResponse = _mapper.Map<OrderResponse>(order);

  if (orderResponse != null)
  {
    foreach (OrderItemResponse orderItemResponse in orderResponse.OrderItems)
    {
      ProductDTO? productDTO = await
_productsMicroserviceClient.GetProductByProductID(orderItemResponse.ProductID);
      if (productDTO != null)
      {
```

```
        _mapper.Map(productDTO, orderItemResponse);
      }
    }

    UserDTO? user = await _usersMicroserviceClient.GetUserByUserID(orderResponse.UserID);
    if (user != null)
    {
      _mapper.Map(user, orderResponse);
    }
  }

  return orderResponse;
}

public async Task<List<OrderResponse?>> GetOrdersByCondition(FilterDefinition<Order> filter)
{
  IEnumerable<Order?> orders = await _ordersRepository.GetOrdersByCondition(filter);
  IEnumerable<OrderResponse?> orderResponses =
_mapper.Map<IEnumerable<OrderResponse>>(orders);

  foreach (OrderResponse? orderResponse in orderResponses)
  {
    if (orderResponse != null)
    {
      foreach (OrderItemResponse orderItemResponse in orderResponse.OrderItems)
      {
        ProductDTO? productDTO = await
_productsMicroserviceClient.GetProductByProductID(orderItemResponse.ProductID);
        if (productDTO != null)
        {
          _mapper.Map(productDTO, orderItemResponse);
        }
```

```
      }

      UserDTO? user = await _usersMicroserviceClient.GetUserByUserID(orderResponse.UserID);
      if (user != null)
      {
        _mapper.Map(user, orderResponse);
      }
     }
    }

    return orderResponses.ToList();
  }

  public async Task<List<OrderResponse?>> GetOrders()
  {
    IEnumerable<Order?> orders = await _ordersRepository.GetOrders();
    IEnumerable<OrderResponse?> orderResponses =
_mapper.Map<IEnumerable<OrderResponse>>(orders);

    foreach (OrderResponse? orderResponse in orderResponses)
    {
      if (orderResponse != null)
      {
        foreach (OrderItemResponse orderItemResponse in orderResponse.OrderItems)
        {
          ProductDTO? productDTO = await
_productsMicroserviceClient.GetProductByProductID(orderItemResponse.ProductID);
          if (productDTO != null)
          {
            _mapper.Map(productDTO, orderItemResponse);
          }
        }
```

```
        UserDTO? user = await _usersMicroserviceClient.GetUserByUserID(orderResponse.UserID);
        if (user != null)
        {
          _mapper.Map(user, orderResponse);
        }
      }
    }

    return orderResponses.ToList();
  }
}
```

Explanation

- **Dependencies**: The constructor injects various dependencies, including repositories, clients, and validators.
- **AddOrder**:
    - Validates the incoming request.
    - Retrieves products and user data from microservices.
    - Maps the request to an Order entity and calculates total prices.
    - Adds the order to the repository and maps the result to a response object.
- **UpdateOrder**:
    - Similar to AddOrder, but updates an existing order.
    - Retrieves and validates order items and user.
    - Updates the order in the repository and returns the updated response.
- **DeleteOrder**:
    - Deletes an order based on its ID.
    - Checks if the order exists before attempting deletion.
- **GetOrderByCondition**:
    - Retrieves a single order based on a filter.
    - Maps the order to a response object, including product and user details.

- **GetOrdersByCondition**:
    - Retrieves a list of orders based on a filter.
    - Maps each order and its details to response objects.
- **GetOrders**:
    - Retrieves all orders.
    - Maps each order to a response object with product and user details.

Code Explanation - Dependency Injection in BusinessLogicLayer

The DependencyInjection class in the BusinessLogicLayer is designed to register services and validators into the dependency injection container. This setup ensures that the business logic layer's dependencies are properly managed and injected throughout the application.

DependencyInjection Class

```
using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using FluentValidation;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.Validators;

using eCommerce.ordersMicroservice.BusinessLogicLayer.Mappers;

using eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts;

using eCommerce.ordersMicroservice.BusinessLogicLayer.Services;


namespace eCommerce.OrdersMicroservice.BusinessLogicLayer;


public static class DependencyInjection

{

  public static IServiceCollection AddBusinessLogicLayer(this IServiceCollection services,
IConfiguration configuration)

  {

    // Register all validators from the assembly containing the OrderAddRequestValidator

    services.AddValidatorsFromAssemblyContaining<OrderAddRequestValidator>();
```

```
    // Register AutoMapper profiles from the assembly containing the
OrderAddRequestToOrderMappingProfile
    services.AddAutoMapper(typeof(OrderAddRequestToOrderMappingProfile).Assembly);


    // Register the OrdersService implementation for the IOrdersService interface
    services.AddScoped<IOrdersService, OrdersService>();


    return services;
  }
}
```

Explanation

1. **AddValidatorsFromAssemblyContaining<OrderAddRequestValidator>()**:
    - This line registers all validators in the assembly where OrderAddRequestValidator is located.
    - FluentValidation uses this to automatically discover and register all validator types, so you don't need to add each validator individually.
2. **AddAutoMapper(typeof(OrderAddRequestToOrderMappingProfile).Assembly)**:
    - This registers AutoMapper profiles from the assembly containing the OrderAddRequestToOrderMappingProfile.
    - AutoMapper is used to map between different object models (e.g., mapping DTOs to entities). By specifying the assembly, AutoMapper will look for any profiles in that assembly.
3. **AddScoped<IOrdersService, OrdersService>()**:
    - Registers the OrdersService class as the implementation for the IOrdersService interface.
    - The Scoped lifetime means that a new instance of OrdersService will be created for each request within the scope (typically a web request).
4. **Return services**:
    - The services object is returned, allowing for method chaining and adding more services or configurations if needed.

Code Explanation - appsettings.json

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings":
  {
    "MongoDB": "mongodb://$MONGO_HOST:$MONGO_PORT"
  }
}
```

This file is used to configure various settings for your ASP.NET Core application. Here's what each section does:

Logging Configuration

```
"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft.AspNetCore": "Warning"
  }
}
```

- **Default**: Sets the default logging level to Information. This means that logs of level Information and higher (like Warning, Error, Critical) will be recorded.
- **Microsoft.AspNetCore**: Sets the logging level for ASP.NET Core to Warning. Only warnings and errors from ASP.NET Core will be logged.

AllowedHosts

"AllowedHosts": "*"

- *****: Allows requests from any host. In production, you might restrict this to specific hosts for security reasons.

ConnectionStrings

"ConnectionStrings": {

"MongoDB": "mongodb://$MONGO_HOST:$MONGO_PORT"

}

- **MongoDB**: Contains the connection string for MongoDB. Replace $MONGO_HOST and $MONGO_PORT with actual values. For example:

"ConnectionStrings": {

"MongoDB": "mongodb://localhost:27017"

}

This connection string tells your application how to connect to the MongoDB database. In practice, use environment variables to inject these values securely.

Code Explanation - Exception Handling Middleware in API layer

This middleware component is designed to handle exceptions that occur during the processing of HTTP requests in your ASP.NET Core application.

ExceptionHandlingMiddleware Class

```
public class ExceptionHandlingMiddleware
{
  private readonly RequestDelegate _next;
  private readonly ILogger<ExceptionHandlingMiddleware> _logger;

  public ExceptionHandlingMiddleware(RequestDelegate next,
ILogger<ExceptionHandlingMiddleware> logger)
  {
    _next = next;
```

```
    _logger = logger;
  }


  public async Task Invoke(HttpContext httpContext)
  {
    try
    {
      await _next(httpContext);
    }
    catch (Exception ex)
    {
      if (ex.InnerException != null)
      {
        _logger.LogError("{ExceptionType} {ExceptionMessage}",
ex.InnerException.GetType().ToString(), ex.InnerException.Message);
      }
      else
      {
        _logger.LogError("{ExceptionType} {ExceptionMessage}", ex.GetType().ToString(), ex.Message);
      }


      httpContext.Response.StatusCode = 500;
      await httpContext.Response.WriteAsJsonAsync(new { Message = ex.Message, Type =
ex.GetType().ToString() });
    }
  }
}
```

- **Purpose**: Catches exceptions that occur during the request processing pipeline.
- **Constructor**: Initializes the middleware with the next request delegate and a logger.
- **Invoke Method**:
    - Wraps the next middleware in a try-catch block.
    - Logs detailed error information including the type and message of the exception.

- Sets the HTTP response status code to 500 (Internal Server Error).
- Returns a JSON response with the error message and type.

Code Explanation - Program.cs in API layer

This file configures and sets up the services and middleware for the ASP.NET Core application.

```
using eCommerce.OrdersMicroservice.DataAccessLayer;
using eCommerce.OrdersMicroservice.BusinessLogicLayer;
using FluentValidation.AspNetCore;
using eCommerce.OrdersMicroservice.API.Middleware;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.HttpClients;

var builder = WebApplication.CreateBuilder(args);

// Add DAL and BLL services
builder.Services.AddDataAccessLayer(builder.Configuration);
builder.Services.AddBusinessLogicLayer(builder.Configuration);

builder.Services.AddControllers();

// FluentValidation
builder.Services.AddFluentValidationAutoValidation();

// Swagger
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

// CORS
builder.Services.AddCors(options => {
  options.AddDefaultPolicy(builder =>
```

```
 {
   builder.WithOrigins("http://localhost:4200")
          .AllowAnyMethod()
          .AllowAnyHeader();
 });
});

// HTTP Clients for microservices
builder.Services.AddHttpClient<UsersMicroserviceClient>(client => {
  client.BaseAddress = new
Uri($"http://{builder.Configuration["UsersMicroserviceName"]}:{builder.Configuration["UsersMicros
ervicePort"]}");
});

builder.Services.AddHttpClient<ProductsMicroserviceClient>(client => {
  client.BaseAddress = new
Uri($"http://{builder.Configuration["ProductsMicroserviceName"]}:{builder.Configuration["Products
MicroservicePort"]}");
});

var app = builder.Build();

// Middleware
app.UseExceptionHandlingMiddleware();
app.UseRouting();

// CORS
app.UseCors();

// Swagger
app.UseSwagger();
app.UseSwaggerUI();
```

```
// Authentication & Authorization
// app.UseHttpsRedirection(); // Uncomment if HTTPS redirection is needed
app.UseAuthentication();
app.UseAuthorization();

// Endpoints
app.MapControllers();

app.Run();
```

Key Components:

1. **Service Registration**:
    - AddDataAccessLayer: Registers the data access layer services.
    - AddBusinessLogicLayer: Registers business logic layer services.
    - AddControllers: Adds services for controllers.
    - AddFluentValidationAutoValidation: Configures FluentValidation.
    - AddEndpointsApiExplorer and AddSwaggerGen: Configures Swagger for API documentation.
2. **CORS Policy**:
    - Allows requests from http://localhost:4200 and permits any method and header.
3. **HTTP Clients**:
    - Configures UsersMicroserviceClient and ProductsMicroserviceClient with base addresses derived from configuration settings.
4. **Middleware Configuration**:
    - UseExceptionHandlingMiddleware: Adds custom exception handling middleware.
    - UseRouting: Adds routing middleware.
    - UseCors: Adds CORS middleware.
    - UseSwagger and UseSwaggerUI: Adds Swagger documentation middleware.
    - UseAuthentication and UseAuthorization: Configures authentication and authorization.
    - MapControllers: Maps controller endpoints.

5. **Application Build and Run**:
    - Builds and runs the web application.

Code Explanation - Orders API Controller in API layer

This controller handles HTTP requests for managing orders within the Orders microservice. It utilizes the IOrdersService interface to perform various operations related to orders.

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;
using Microsoft.AspNetCore.Mvc;
using MongoDB.Driver;

namespace OrdersMicroservice.API.ApiControllers;

[Route("api/[controller]")]
[ApiController]
public class OrdersController : ControllerBase
{
   private readonly IOrdersService _ordersService;

   public OrdersController(IOrdersService ordersService)
   {
      _ordersService = ordersService;
   }

   // GET: /api/Orders
   [HttpGet]
   public async Task<IEnumerable<OrderResponse?>> Get()
   {
```

```
        List<OrderResponse?> orders = await _ordersService.GetOrders();
        return orders;
    }

    // GET: /api/Orders/search/orderid/{orderID}
    [HttpGet("search/orderid/{orderID}")]
    public async Task<OrderResponse?> GetOrderByOrderID(Guid orderID)
    {
        FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, orderID);
        OrderResponse? order = await _ordersService.GetOrderByCondition(filter);
        return order;
    }

    // GET: /api/Orders/search/productid/{productID}
    [HttpGet("search/productid/{productID}")]
    public async Task<IEnumerable<OrderResponse?>> GetOrdersByProductID(Guid productID)
    {
        FilterDefinition<Order> filter = Builders<Order>.Filter.ElemMatch(temp => temp.OrderItems,
            Builders<OrderItem>.Filter.Eq(tempProduct => tempProduct.ProductID, productID));
        List<OrderResponse?> orders = await _ordersService.GetOrdersByCondition(filter);
        return orders;
    }

    // GET: /api/Orders/search/orderDate/{orderDate}
    [HttpGet("/search/orderDate/{orderDate}")]
    public async Task<IEnumerable<OrderResponse?>> GetOrdersByOrderDate(DateTime orderDate)
    {
        FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp =>
temp.OrderDate.ToString("yyyy-MM-dd"), orderDate.ToString("yyyy-MM-dd"));
        List<OrderResponse?> orders = await _ordersService.GetOrdersByCondition(filter);
        return orders;
```

```
}

// POST api/Orders
[HttpPost]
public async Task<IActionResult> Post(OrderAddRequest orderAddRequest)
{
    if (orderAddRequest == null)
    {
        return BadRequest("Invalid order data");
    }

    OrderResponse? orderResponse = await _ordersService.AddOrder(orderAddRequest);

    if (orderResponse == null)
    {
        return Problem("Error in adding order");
    }

    return Created($"api/Orders/search/orderid/{orderResponse?.OrderID}", orderResponse);
}

// PUT api/Orders/{orderID}
[HttpPut("{orderID}")]
public async Task<IActionResult> Put(Guid orderID, OrderUpdateRequest orderUpdateRequest)
{
    if (orderUpdateRequest == null)
    {
        return BadRequest("Invalid order data");
    }

    if (orderID != orderUpdateRequest.OrderID)
```

```
        {
            return BadRequest("OrderID in the URL doesn't match with the OrderID in the Request body");
        }

        OrderResponse? orderResponse = await _ordersService.UpdateOrder(orderUpdateRequest);

        if (orderResponse == null)
        {
            return Problem("Error in updating order");
        }

        return Ok(orderResponse);
    }

    // DELETE api/Orders/{orderID}
    [HttpDelete("{orderID}")]
    public async Task<IActionResult> Delete(Guid orderID)
    {
        if (orderID == Guid.Empty)
        {
            return BadRequest("Invalid order ID");
        }

        bool isDeleted = await _ordersService.DeleteOrder(orderID);

        if (!isDeleted)
        {
            return Problem("Error in deleting order");
        }
```

```
        return Ok(isDeleted);
    }
}
```

Key Endpoints:

1. **GET /api/Orders**: Retrieves a list of all orders.
2. **GET /api/Orders/search/orderid/{orderID}**: Retrieves an order by its OrderID.
3. **GET /api/Orders/search/productid/{productID}**: Retrieves orders containing a specific ProductID.
4. **GET /api/Orders/search/orderDate/{orderDate}**: Retrieves orders by a specific order date.
5. **POST /api/Orders**: Adds a new order. Expects an OrderAddRequest object in the request body.
6. **PUT /api/Orders/{orderID}**: Updates an existing order. Expects an OrderUpdateRequest object in the request body.
7. **DELETE /api/Orders/{orderID}**: Deletes an order by its OrderID.

Error Handling:

- **400 Bad Request**: Returns if the request body is invalid or OrderID mismatch.
- **500 Internal Server Error**: Returns if there's an issue with adding, updating, or deleting an order.

Detailed Explanation of OrdersController

The OrdersController is part of the API layer in an ASP.NET Core application and manages HTTP requests related to orders. It utilizes the IOrdersService interface to perform operations on orders. Here's a breakdown of each part of the controller:

Namespace and Using Directives

```
using eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO;
using eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts;
using eCommerce.OrdersMicroservice.DataAccessLayer.Entities;
using Microsoft.AspNetCore.Mvc;
```

using MongoDB.Driver;

- **eCommerce.OrdersMicroservice.BusinessLogicLayer.DTO**: Contains data transfer objects (DTOs) used for communication between the API and business logic layers.
- **eCommerce.OrdersMicroservice.BusinessLogicLayer.ServiceContracts**: Contains service interfaces that define business logic operations.
- **eCommerce.OrdersMicroservice.DataAccessLayer.Entities**: Contains entity classes representing the data structure in MongoDB.
- **Microsoft.AspNetCore.Mvc**: Provides the base classes for MVC controllers and actions.
- **MongoDB.Driver**: Provides MongoDB-specific features, such as filters and queries.

Controller Definition

```
namespace OrdersMicroservice.API.ApiControllers;


[Route("api/[controller]")]
[ApiController]
public class OrdersController : ControllerBase
{
   private readonly IOrdersService _ordersService;


   public OrdersController(IOrdersService ordersService)
   {
      _ordersService = ordersService;
   }
```

- **[Route("api/[controller]")]**: Defines the base route for all actions in this controller. [controller] is replaced with the controller name (in this case, Orders).
- **[ApiController]**: Makes the controller an API controller with automatic model state validation and binding source parameter inference.

GET: /api/Orders

```
[HttpGet]
public async Task<IEnumerable<OrderResponse?>> Get()
{
   List<OrderResponse?> orders = await _ordersService.GetOrders();
```

```
    return orders;
}
```

- **[HttpGet]**: Specifies that this action responds to HTTP GET requests.
- **Get()**: Retrieves all orders from the data source.
- **_ordersService.GetOrders()**: Calls the business logic layer to get a list of orders.
- **IEnumerable<OrderResponse?>**: Returns a collection of OrderResponse objects.

GET: /api/Orders/search/orderid/{orderID}

```
[HttpGet("search/orderid/{orderID}")]
public async Task<OrderResponse?> GetOrderByOrderID(Guid orderID)
{
    FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderID, orderID);
    OrderResponse? order = await _ordersService.GetOrderByCondition(filter);
    return order;
}
```

- **[HttpGet("search/orderid/{orderID}")]**: Routes to this action when the URL matches /api/Orders/search/orderid/{orderID}.
- **GetOrderByOrderID(Guid orderID)**: Retrieves a specific order based on the provided orderID.
- **FilterDefinition<Order> filter**: Creates a MongoDB filter to match the OrderID.
- **_ordersService.GetOrderByCondition(filter)**: Calls the service layer to get the order matching the filter.

GET: /api/Orders/search/productid/{productID}

```
[HttpGet("search/productid/{productID}")]
public async Task<IEnumerable<OrderResponse?>> GetOrdersByProductID(Guid productID)
{
    FilterDefinition<Order> filter = Builders<Order>.Filter.ElemMatch(temp => temp.OrderItems,
        Builders<OrderItem>.Filter.Eq(tempProduct => tempProduct.ProductID, productID));
    List<OrderResponse?> orders = await _ordersService.GetOrdersByCondition(filter);
    return orders;
}
```

- **[HttpGet("search/productid/{productID}")]**: Routes to this action when the URL matches /api/Orders/search/productid/{productID}.
- **GetOrdersByProductID(Guid productID)**: Retrieves orders containing a specific product.
- **FilterDefinition<Order> filter**: Creates a filter to match orders that have an item with the given ProductID.
- **_ordersService.GetOrdersByCondition(filter)**: Calls the service layer to get orders matching the filter.

GET: /api/Orders/search/orderDate/{orderDate}

```
[HttpGet("/search/orderDate/{orderDate}")]
public async Task<IEnumerable<OrderResponse?>> GetOrdersByOrderDate(DateTime orderDate)
{
  FilterDefinition<Order> filter = Builders<Order>.Filter.Eq(temp => temp.OrderDate.ToString("yyyy-MM-dd"), orderDate.ToString("yyyy-MM-dd"));
  List<OrderResponse?> orders = await _ordersService.GetOrdersByCondition(filter);
  return orders;
}
```

- **[HttpGet("/search/orderDate/{orderDate}")]**: Routes to this action when the URL matches /search/orderDate/{orderDate}.
- **GetOrdersByOrderDate(DateTime orderDate)**: Retrieves orders based on the order date.
- **FilterDefinition<Order> filter**: Creates a filter to match orders with the specified orderDate.
- **_ordersService.GetOrdersByCondition(filter)**: Calls the service layer to get orders matching the filter.

POST: api/Orders

```
[HttpPost]
public async Task<IActionResult> Post(OrderAddRequest orderAddRequest)
{
  if (orderAddRequest == null)
  {
    return BadRequest("Invalid order data");
  }
```

```
    OrderResponse? orderResponse = await _ordersService.AddOrder(orderAddRequest);


    if (orderResponse == null)
    {
        return Problem("Error in adding order");
    }


    return Created($"api/Orders/search/orderid/{orderResponse?.OrderID}", orderResponse);
}
```

- **[HttpPost]**: Specifies that this action responds to HTTP POST requests.
- **Post(OrderAddRequest orderAddRequest)**: Adds a new order. Expects an OrderAddRequest object in the request body.
- **BadRequest("Invalid order data")**: Returns if the request body is null.
- **_ordersService.AddOrder(orderAddRequest)**: Calls the service layer to add the new order.
- **Created($"api/Orders/search/orderid/{orderResponse?.OrderID}", orderResponse)**: Returns a 201 Created response with the location of the newly created order and the order details.

PUT: api/Orders/{orderID}

```
[HttpPut("{orderID}")]
public async Task<IActionResult> Put(Guid orderID, OrderUpdateRequest orderUpdateRequest)
{
    if (orderUpdateRequest == null)
    {
        return BadRequest("Invalid order data");
    }


    if (orderID != orderUpdateRequest.OrderID)
    {
        return BadRequest("OrderID in the URL doesn't match with the OrderID in the Request body");
    }
```

```
    OrderResponse? orderResponse = await _ordersService.UpdateOrder(orderUpdateRequest);

    if (orderResponse == null)
    {
        return Problem("Error in updating order");
    }

    return Ok(orderResponse);
}
```

- **[HttpPut("{orderID}")]**: Routes to this action when the URL matches /api/Orders/{orderID}.
- **Put(Guid orderID, OrderUpdateRequest orderUpdateRequest)**: Updates an existing order. Expects an OrderUpdateRequest object in the request body.
- **BadRequest("Invalid order data")**: Returns if the request body is null.
- **BadRequest("OrderID in the URL doesn't match with the OrderID in the Request body")**: Returns if the OrderID in the URL does not match the one in the request body.
- **_ordersService.UpdateOrder(orderUpdateRequest)**: Calls the service layer to update the order.
- **Ok(orderResponse)**: Returns a 200 OK response with the updated order details.

DELETE: api/Orders/{orderID}

```
[HttpDelete("{orderID}")]
public async Task<IActionResult> Delete(Guid orderID)
{
    if (orderID == Guid.Empty)
    {
        return BadRequest("Invalid order ID");
    }

    bool isDeleted = await _ordersService.DeleteOrder(orderID);

    if (!isDeleted)
    {
```

```
        return Problem("Error in deleting order");
    }


    return Ok(isDeleted);
}
```

- **[HttpDelete("{orderID}")]**: Routes to this action when the URL matches /api/Orders/{orderID}.
- **Delete(Guid orderID)**: Deletes an order by its OrderID.
- **BadRequest("Invalid order ID")**: Returns if the orderID is empty.
- **_ordersService.DeleteOrder(orderID)**: Calls the service layer to delete the order.
- **Ok(isDeleted)**: Returns a 200 OK response indicating whether the order was successfully deleted.