Part 5: Customer Management

Objective: We'll implement customer management in our e-commerce API, using DTOs for cleaner data handling, AutoMapper for mapping, and improved validation techniques.

1. Create the Customer Entity

- File: Models/Customer.cs

```
 namespace ECommerceAPI.Models
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Email { get; set; }
        public string PhoneNumber { get; set; }
        public string Address { get; set; }
    }
}
```

- Explanation:
  - The Customer model remains simple since it doesn't have relationships in this step. However, you might want to consider adding validation attributes here for data integrity, e.g., [EmailAddress] for email validation.

2. Update the Database Context

- File: Data/ECommerceDbContext.cs

```
public DbSet<Customer> Customers { get; set; }
```

- Explanation:
  - Adding the Customers DbSet to manage customer data in the database.

Run a new migration and update the database:

```
dotnet ef migrations add AddCustomerEntity
dotnet ef database update
```

3. Create DTOs for Customers

- Folder: DTOs
- CustomerCreateDTO.cs:

```
namespace ECommerceAPI.DTOs
{
   public class CustomerCreateDTO
   {
      [Required]
      public string FirstName { get; set; }
      [Required]
      public string LastName { get; set; }
      [Required, EmailAddress]
      public string Email { get; set; }
      [Phone]
      public string PhoneNumber { get; set; }
      public string Address { get; set; }
   }
}
```

- CustomerDTO.cs:

```
namespace ECommerceAPI.DTOs
{
   public class CustomerDTO
   {
      public int Id { get; set; }
      public string FirstName { get; set; }
      public string LastName { get; set; }
      public string Email { get; set; }
      public string PhoneNumber { get; set; }
      public string Address { get; set; }
   }
}
```

- Explanation:
  - DTOs help in controlling the data that goes in and out of your API, providing a layer for validation and ensuring you're not exposing unnecessary data.

4. Configure AutoMapper for Customers

- File: Profiles/CustomerProfile.cs

```
using AutoMapper;
using ECommerceAPI.Models;
using ECommerceAPI.DTOs;

namespace ECommerceAPI.Profiles
{
    public class CustomerProfile : Profile
    {
        public CustomerProfile()
        {
            CreateMap<CustomerCreateDTO, Customer>();
            CreateMap<Customer, CustomerDTO>();
        }
    }
}
```

- Explanation:
    - This profile maps between CustomerCreateDTO for creating customers and CustomerDTO for returning customer data.

5. Create the Customers Controller

- File: Controllers/CustomersController.cs

```
using ECommerceAPI.Data;
using ECommerceAPI.Models;
using ECommerceAPI.DTOs;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using AutoMapper;

namespace ECommerceAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    [Authorize]
    public class CustomersController : ControllerBase
    {
        private readonly ECommerceDbContext _context;
        private readonly IMapper _mapper;

        public CustomersController(ECommerceDbContext context, IMapper mapper)
        {
            _context = context;
            _mapper = mapper;
        }

        // GET: api/Customers
        [HttpGet]
        public async Task<ActionResult<IEnumerable<CustomerDTO>>> GetCustomers()
        {
            var customers = await _context.Customers.ToListAsync();
            return Ok(_mapper.Map<IEnumerable<CustomerDTO>>(customers));
        }

        // GET: api/Customers/5
        [HttpGet("{id}")]
        public async Task<ActionResult<CustomerDTO>> GetCustomer(int id)
        {
            var customer = await _context.Customers.FindAsync(id);
            if (customer == null)
            {
                return NotFound();
            }
            return _mapper.Map<CustomerDTO>(customer);
        }

        // POST: api/Customers
        [HttpPost]
        public async Task<ActionResult<CustomerDTO>> PostCustomer(CustomerCreateDTO
customerDto)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }

            var customer = _mapper.Map<Customer>(customerDto);
            _context.Customers.Add(customer);
            await _context.SaveChangesAsync();

            return CreatedAtAction(nameof(GetCustomer), new { id = customer.Id },
_mapper.Map<CustomerDTO>(customer));
        }

        // PUT: api/Customers/5
```

```
[HttpPut("{id}")]
public async Task<IActionResult> PutCustomer(int id, CustomerCreateDTO customerDto)
{
    if (id != customerDto.Id)
    {
        return BadRequest();
    }

    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var customer = await _context.Customers.FindAsync(id);
    if (customer == null)
    {
        return NotFound();
    }

    _mapper.Map(customerDto, customer);
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!CustomerExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}

// DELETE: api/Customers/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteCustomer(int id)
{
    var customer = await _context.Customers.FindAsync(id);
```

```
            if (customer == null)
            {
                return NotFound();
            }

            _context.Customers.Remove(customer);
            await _context.SaveChangesAsync();

            return NoContent();
        }

        private bool CustomerExists(int id)
        {
            return _context.Customers.Any(e => e.Id == id);
        }
    }
}
```

- Explanation:
  - IMapper: Used in the controller for mapping between DTOs and entities.
  - ModelState.IsValid: This checks data annotations on your DTO for validation, replacing the custom IsValidEmail method.
  - Mapping: AutoMapper is used consistently for converting between DTOs and entities, which simplifies the code and ensures consistency.

6. Validate Customer Data

- Explanation:
  - Validation is now done through data annotations in the DTOs, which are automatically checked by ASP.NET Core when binding the model. No need for custom methods like IsValidEmail.

7. Test the Endpoints

- Explanation:
  - Testing steps remain similar, but now you're testing with DTOs. Ensure that the validation errors from data annotations are correctly returned when invalid data is sent.

8. Extend Product and Order Integration (Optional)

- Future Planning:

    - When you integrate orders, consider adding a virtual ICollection<Order> to Customer for the relationship. Also, think about using DTOs to manage this relationship in your API responses to avoid circular references.

1. Review DTO Definitions:

- In your DTO for creating a customer (CustomerCreateDTO), you shouldn't need or include an Id since this is typically auto-generated or set by the database when a new customer is created. Here's how it should look:

```
public class CustomerCreateDTO
{
   [Required]
   public string FirstName { get; set; }
   [Required]
   public string LastName { get; set; }
   [Required, EmailAddress]
   public string Email { get; set; }
   [Phone]
   public string PhoneNumber { get; set; }
   public string Address { get; set; }
}
```

2. Modify the PutCustomer Method:

- The PUT operation expects you to update an existing entity, so you pass the id through the route, not in the DTO. Here's how you should adjust the PutCustomer method:

```
[HttpPut("{id}")]
public async Task<IActionResult> PutCustomer(int id, CustomerCreateDTO customerDto)
{
   if (!ModelState.IsValid)
   {
      return BadRequest(ModelState);
   }

   var customer = await _context.Customers.FindAsync(id);
   if (customer == null)
   {
      return NotFound();
   }

   _mapper.Map(customerDto, customer); // Map from DTO to existing customer entity
```

```
    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!CustomerExists(id))
        {
            return NotFound();
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}
```

- Explanation:
  - We removed the id != customerDto.Id check because customerDto does not have an Id property. Instead, we use the id from the route to find the existing customer and then map the properties from CustomerCreateDTO to this customer entity.

3. Mapping in AutoMapper:

- Your AutoMapper profile for customers should remain as is:

```
public class CustomerProfile : Profile
{
    public CustomerProfile()
    {
        CreateMap<CustomerCreateDTO, Customer>();
        CreateMap<Customer, CustomerDTO>();
    }
}
```

- This setup will correctly map from CustomerCreateDTO to Customer without needing to deal with an Id in the DTO for creation or updates.

These changes should resolve the error by ensuring that you're not trying to access or compare an Id property on the CustomerCreateDTO where it doesn't exist. Remember, when updating, the Id comes from the route parameter, not from the DTO.

2. Authentication:

- Since your API uses authorization, you'll need to authenticate first:
    - If using JWT:
        - First, get a JWT token by making a POST request to your login endpoint (assuming you have one). Include username and password in the body.
        - Copy the token from the response.
        - In Postman, go to the Authorization tab for each request, select "Bearer Token" and paste your JWT token.

3. Testing Endpoints:

- GET /api/Customers
    - Method: GET
    - URL: http://localhost:<yourPort>/api/Customers
    - Expected: A list of all customers or an empty list if none exist.
    - Test: Verify you get an array of customers with all expected fields.
- GET /api/Customers/{id}
    - Method: GET
    - URL: http://localhost:<yourPort>/api/Customers/1 (replace 1 with an actual ID if known)
    - Expected: Details of the customer with the specified ID or a 404 if not found.
    - Test: Try with both existing and non-existing IDs.
- POST /api/Customers
    - Method: POST
    - URL: http://localhost:<yourPort>/api/Customers
    - Body:

```json
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@example.com",
  "phoneNumber": "123-456-7890",
  "address": "123 Main St, Springfield, USA"
}
```

- Expected: A 201 Created response with a location header pointing to the new customer's URI and the customer data in the response body.
- Test:

    - Create a new customer, check if the response includes the new customer's details.
    - Try sending invalid data (like an invalid email) to see if validation works.

PUT /api/Customers/{id}

- Method: PUT
- URL: http://localhost:<yourPort>/api/Customers/1
- Body: (Update with new or changed data)

```
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "updated.email@example.com",
  "phoneNumber": "987-654-3210",
  "address": "456 Elm St, Springfield, USA"
}
```

    - Expected: A 204 No Content response if successful.
    - Test:
        - Update an existing customer, then retrieve to confirm changes.
        - Try updating a non-existent customer to check for a 404 response.
- DELETE /api/Customers/{id}
    - Method: DELETE
    - URL: http://localhost:<yourPort>/api/Customers/1
    - Expected: A 204 No Content response if the customer was deleted.
    - Test:
        - Delete a customer, then try to GET it to ensure it returns a 404.
        - Try to delete a non-existent customer to check the error handling.