Part 6: Order Management

In this part, we'll add functionality to manage orders. Orders will include multiple items, and we'll track the relationship between `Orders`, `OrderItems`, `Products`, and `Customers`.

---

1. Create the Order and OrderItem Entities

- Add a new file `Models/Order.cs`:

```
using System;

namespace ECommerceAPI.Models
{
        public class Order
        {
        public int Id { get; set; }
        public int CustomerId { get; set; }
        public Customer Customer { get; set; }
        public DateTime OrderDate { get; set; }
        public decimal TotalAmount { get; set; }
        public ICollection<OrderItem> OrderItems { get; set; }
        }
}
```

Add a new file `Models/OrderItem.cs`:

```
namespace ECommerceAPI.Models
{
        public class OrderItem
        {
        public int Id { get; set; }
        public int OrderId { get; set; }
        public Order Order { get; set; }
        public int ProductId { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
        public decimal Price { get; set; }
        }
}
```

2. Update the Database Context

- Add `Orders` and `OrderItems` DbSets to `ECommerceDbContext`:

```
public DbSet<Order> Orders { get; set; }
public DbSet<OrderItem> OrderItems { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
	modelBuilder.Entity<Order>()
	.HasMany(o => o.OrderItems)
	.WithOne(oi => oi.Order)
	.HasForeignKey(oi => oi.OrderId);

	modelBuilder.Entity<OrderItem>()
	.HasOne(oi => oi.Product)
	.WithMany()
	.HasForeignKey(oi => oi.ProductId);

	base.OnModelCreating(modelBuilder);
}
```

Run a new migration and update the database:

```
dotnet ef migrations add AddOrderEntities
dotnet ef database update
```

3. Create the Orders Controller

- Add a new controller `Controllers/OrdersController.cs`:

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

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

```
}

public OrdersController(ECommerceDbContext context)
{
_context = context;
}

// GET: api/Orders
[HttpGet]
public async Task<ActionResult<IEnumerable<Order>>> GetOrders()
{
return await _context.Orders
        .Include(o => o.Customer)
        .Include(o => o.OrderItems)
        .ThenInclude(oi => oi.Product)
        .ToListAsync();
}

// GET: api/Orders/5
[HttpGet("{id}")]
public async Task<ActionResult<Order>> GetOrder(int id)
{
var order = await _context.Orders
        .Include(o => o.Customer)
        .Include(o => o.OrderItems)
        .ThenInclude(oi => oi.Product)
        .FirstOrDefaultAsync(o => o.Id == id);

if (order == null)
{
        return NotFound();
}

return order;
}

// POST: api/Orders
[HttpPost]
public async Task<ActionResult<Order>> PostOrder(Order order)
{
// Validate customer existence
var customer = await _context.Customers.FindAsync(order.CustomerId);
if (customer == null)
{
        return BadRequest("Customer not found.");
}
```

```
// Validate product stock and calculate total
decimal totalAmount = 0;
foreach (var item in order.OrderItems)
{
        var product = await _context.Products.FindAsync(item.ProductId);
        if (product == null || product.Stock < item.Quantity)
        {
        return BadRequest($"Insufficient stock for product ID {item.ProductId}");
        }

        item.Price = product.Price * item.Quantity;
        totalAmount += item.Price;

        // Deduct stock
        product.Stock -= item.Quantity;
}

order.OrderDate = DateTime.UtcNow;
order.TotalAmount = totalAmount;

_context.Orders.Add(order);
await _context.SaveChangesAsync();

return CreatedAtAction(nameof(GetOrder), new { id = order.Id }, order);
}

// PUT: api/Orders/5
[HttpPut("{id}")]
public async Task<IActionResult> PutOrder(int id, Order order)
{
if (id != order.Id)
{
        return BadRequest();
}

_context.Entry(order).State = EntityState.Modified;

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

return NoContent();
}

// DELETE: api/Orders/5
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteOrder(int id)
{
var order = await _context.Orders
        .Include(o => o.OrderItems)
        .FirstOrDefaultAsync(o => o.Id == id);
if (order == null)
{
        return NotFound();
}

// Return stock for deleted order items
foreach (var item in order.OrderItems)
{
        var product = await _context.Products.FindAsync(item.ProductId);
        if (product != null)
        {
        product.Stock += item.Quantity;
        }
}

_context.Orders.Remove(order);
await _context.SaveChangesAsync();

return NoContent();
}

private bool OrderExists(int id)
{
return _context.Orders.Any(o => o.Id == id);
```

```
        }
        }
}
```

4. Test the Endpoints

Use a tool like Postman or Swagger to test the following endpoints:

1. **Get all orders**:
   - `GET /api/Orders`
2. **Get an order by ID**:
   - `GET /api/Orders/{id}`
3. **Create a new order**:
   - `POST /api/Orders`
   - Body:

```
{
  "customerId": 1,
  "orderItems": [
        {
        "productId": 1,
        "quantity": 2
        },
        {
        "productId": 2,
        "quantity": 1
        }
  ]
}
```

Update an order (optional):

- `PUT /api/Orders/{id}`
- Body:

```
{
  "id": 1,
  "customerId": 1,
  "orderDate": "2024-11-30T00:00:00",
  "totalAmount": 100.00,
  "orderItems": []
}
```

Delete an order:

- `DELETE /api/Orders/{id}`

5. Extend Functionality (Optional)

- **Order Status**: Add a status field to track order processing (e.g., Pending, Shipped, Delivered).
- **Order Filtering**: Allow filtering by customer, date range, or status.
- **Payment Integration**: Integrate a payment gateway to handle payments during order creation.

Part 7: Search and Filtering

In this part, we'll implement search and filtering functionality for products. This allows users to find products by name, category, price range, and other criteria.

---

1. Update the Products Controller

Modify the `ProductsController` to include search and filtering endpoints.

- Add a `SearchProducts` action to handle query parameters:

```
using Microsoft.AspNetCore.Authorization;


[Authorize]

[ApiController]

[Route("api/[controller]")]

public class ProductsController : ControllerBase
```

```
{

      private readonly ECommerceDbContext _context;


      public ProductsController(ECommerceDbContext context)

      {

      _context = context;

      }


      // GET: api/Products

      [HttpGet]

      public async Task<ActionResult<IEnumerable<Product>>>
GetProducts()

      {

      return await _context.Products.Include(p =>
p.Category).ToListAsync();

      }


      // GET: api/Products/search

      [HttpGet("search")]

      public async Task<ActionResult<IEnumerable<Product>>>
SearchProducts(

      [FromQuery] string? name,

      [FromQuery] int? categoryId,

      [FromQuery] decimal? minPrice,
      [FromQuery] decimal? maxPrice)

      {

      var query = _context.Products.AsQueryable();


      // Filter by name

      if (!string.IsNullOrEmpty(name))

      {

           query = query.Where(p => p.Name.Contains(name));

      }


      // Filter by category

      if (categoryId.HasValue)

      {

           query = query.Where(p => p.CategoryId == categoryId);

      }


      // Filter by price range

      if (minPrice.HasValue)

      {

           query = query.Where(p => p.Price >= minPrice.Value);

      }

      if (maxPrice.HasValue)

      {
```

```
            query = query.Where(p => p.Price <= maxPrice.Value);

        }


        // Include category information

        var result = await query.Include(p => p.Category).ToListAsync();

        return Ok(result);

        }

}
```

2. Test the Search Endpoint

Use a tool like Postman or Swagger to test the following scenarios:

1. **Search by name**:
   - `GET /api/Products/search?name=Smartphone`
2. **Filter by category**:
   - `GET /api/Products/search?categoryId=1`
3. **Filter by price range**:
   - `GET /api/Products/search?minPrice=100&maxPrice=500`
4. **Combine filters**:
   - `GET /api/Products/search?name=Phone&minPrice=100&maxPrice=1000&categoryId=1`

3. Add Pagination

Enhance the search functionality with pagination.

- Update the `SearchProducts` method:

```
[HttpGet("search")]
```

```
public async Task<ActionResult<IEnumerable<Product>>> SearchProducts(
      [FromQuery] string? name,
      [FromQuery] int? categoryId,
      [FromQuery] decimal? minPrice,
      [FromQuery] decimal? maxPrice,
      [FromQuery] int page = 1,
      [FromQuery] int pageSize = 10)
{
      var query = _context.Products.AsQueryable();

      // Apply filters
      if (!string.IsNullOrEmpty(name))
      {
      query = query.Where(p => p.Name.Contains(name));
      }
      if (categoryId.HasValue)
      {
      query = query.Where(p => p.CategoryId == categoryId);
      }
      if (minPrice.HasValue)
      {
      query = query.Where(p => p.Price >= minPrice.Value);
      }
```

```
    if (maxPrice.HasValue)

    {

    query = query.Where(p => p.Price <= maxPrice.Value);

    }


    // Include category information

    var totalItems = await query.CountAsync();

    var products = await query.Include(p => p.Category)

    .Skip((page - 1) * pageSize)

    .Take(pageSize)

    .ToListAsync();


    return Ok(new

    {

    TotalItems = totalItems,

    Page = page,

    PageSize = pageSize,

    Products = products

    });

}
```

4. Test Pagination

Test the pagination feature:

1. **Page 1 with 5 items per page**:
    - `GET /api/Products/search?page=1&pageSize=5`
2. **Page 2 with 10 items per page**:
    - `GET /api/Products/search?page=2&pageSize=10`

5. Extend Filtering (Optional)

- Add more filters, such as stock availability or sorting (e.g., price ascending/descending).
- Update the `SearchProducts` method to include sorting:

```
[FromQuery] string? sortBy = null,

[FromQuery] bool ascending = true
```

Modify the query to apply sorting:

```
if (!string.IsNullOrEmpty(sortBy))

{

      query = ascending

      ? query.OrderBy(p => EF.Property<object>(p, sortBy))

      : query.OrderByDescending(p => EF.Property<object>(p, sortBy));

}
```

6. Test Extended Filters

1. **Sort by price ascending**:
    - `GET /api/Products/search?sortBy=Price&ascending=true`
2. **Sort by name descending**:
    - `GET /api/Products/search?sortBy=Name&ascending=false`

Part 8: File Upload for Product Images

In this part, we'll implement functionality to upload images for products. Uploaded images will be stored in the server's file system, and the product record will include a URL pointing to the image.

---

1. Create a Folder to Store Images

- Ensure the application has a directory for storing uploaded images. This can be created at runtime if it doesn't exist.

---

2. Update the Product Entity

Add a property for the image URL in Product.cs:

```
public string? ImageUrl { get; set; }
```

Run a new migration and update the database:

```
dotnet ef migrations add AddImageUrlToProduct

dotnet ef database update
```

3. Add File Upload Logic in ProductsController

Modify ProductsController to include an endpoint for uploading images.

```
using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Hosting;

using System.IO;
```

```
[HttpPost("{id}/upload-image")]

public async Task<IActionResult> UploadImage(int id, IFormFile file)

{

      var product = await _context.Products.FindAsync(id);

      if (product == null)

      {

      return NotFound("Product not found.");

      }


      if (file == null || file.Length == 0)

      {

      return BadRequest("Invalid file.");

      }


      // Ensure the images directory exists

      var imagePath = Path.Combine("wwwroot", "images");

      if (!Directory.Exists(imagePath))

      {

      Directory.CreateDirectory(imagePath);

      }


      // Generate a unique file name and save the file
```

```
    var fileName =
$"{Guid.NewGuid()}_{Path.GetFileName(file.FileName)}";

    var filePath = Path.Combine(imagePath, fileName);


    using (var stream = new FileStream(filePath, FileMode.Create))

    {

    await file.CopyToAsync(stream);

    }


    // Update the product's ImageUrl

    product.ImageUrl = $"/images/{fileName}";

    _context.Entry(product).State = EntityState.Modified;

    await _context.SaveChangesAsync();


    return Ok(new { ImageUrl = product.ImageUrl });

}
```

4. Configure Static File Serving

To serve uploaded images, configure static file serving in Program.cs:

```
app.UseStaticFiles();
```

5. Test the File Upload Endpoint

Use a tool like Postman to test the image upload feature:

1. **Upload an image**:
    - Endpoint: `POST /api/Products/{id}/upload-image`
    - Form-data:
        - Key: `file`
        - Value: (Select an image file)
    - Response:

```
{

  "imageUrl": "/images/unique-file-name.jpg"

}
```

Access the uploaded image:

- Open the returned `ImageUrl` in your browser:
    - Example: `http://localhost:5000/images/unique-file-name.jpg`

6. Update the Product Retrieval Logic

Ensure the `ImageUrl` is included when fetching product details:

```
[HttpGet("{id}")]

public async Task<ActionResult<Product>> GetProduct(int id)

{

      var product = await _context.Products.Include(p =>
p.Category).FirstOrDefaultAsync(p => p.Id == id);



      if (product == null)

      {

      return NotFound();

      }
```

```
    return product;
}
```

7. Enhance Validation and Error Handling (Optional)

- **Validate file types**: Ensure only image files are uploaded.

```
var allowedExtensions = new[] { ".jpg", ".jpeg", ".png", ".gif" };
var fileExtension = Path.GetExtension(file.FileName);
if (!allowedExtensions.Contains(fileExtension.ToLower()))
{
    return BadRequest("Invalid file type. Only JPG, JPEG, PNG, and
GIF are allowed.");
}
```

Restrict file size: Limit the file size to prevent excessively large uploads.

8. Cleanup Logic for Image Replacement (Optional)

When updating an image, delete the old image file to free up storage:

```
if (!string.IsNullOrEmpty(product.ImageUrl))
{
    var oldImagePath = Path.Combine("wwwroot",
product.ImageUrl.TrimStart('/'));
    if (System.IO.File.Exists(oldImagePath))
    {
    System.IO.File.Delete(oldImagePath);
    }
}
```

9. Future Enhancements

- **Cloud Storage**: Store images in cloud services like AWS S3, Azure Blob Storage, or Google Cloud Storage instead of the server file system.
- **Image Optimization**: Automatically resize or compress uploaded images for better performance.

Part 9: Payment Integration

In this part, we'll implement a basic payment integration for the e-commerce API using a payment gateway like Stripe. The goal is to allow users to pay for their orders and track payment status.

---

1. Set Up a Stripe Account

1. Go to Stripe and create an account.
2. Obtain your **API Secret Key** from the Stripe Dashboard under Developers > API Keys.

---

2. Install Stripe SDK

Install the official Stripe library:

```
dotnet add package Stripe.net
```

3. Add Payment Integration Settings

- Add `PaymentSettings` in a new file `Helpers/PaymentSettings.cs`:

```
namespace ECommerceAPI.Helpers

{

      public class PaymentSettings

      {

      public string SecretKey { get; set; }

      }

}
```

Add Stripe settings to appsettings.json:

```
"PaymentSettings": {

  "SecretKey": "sk_test_your_secret_key"

}
```

Register PaymentSettings in Program.cs:

```
using ECommerceAPI.Helpers;


builder.Services.Configure<PaymentSettings>(builder.Configuration.GetSection("PaymentSettings"));
```

4. Update the Order Entity

Add a PaymentStatus property to track the status of an order's payment:

```
public string PaymentStatus { get; set; } = "Pending"; // Default status
```

Run a migration to update the database:

```
dotnet ef migrations add AddPaymentStatusToOrder

dotnet ef database update
```

5. Create a Payments Controller

- Add a new controller Controllers/PaymentsController.cs:

```
using ECommerceAPI.Data;

using ECommerceAPI.Helpers;

using ECommerceAPI.Models;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Options;

using Stripe;

using System.Threading.Tasks;


namespace ECommerceAPI.Controllers

{

     [Route("api/[controller]")]

     [ApiController]

     public class PaymentsController : ControllerBase

     {

     private readonly ECommerceDbContext _context;
```

```
        private readonly PaymentSettings _paymentSettings;


        public PaymentsController(ECommerceDbContext context,
IOptions<PaymentSettings> paymentSettings)

        {

            _context = context;

            _paymentSettings = paymentSettings.Value;


            StripeConfiguration.ApiKey = _paymentSettings.SecretKey;

        }


        // POST: api/Payments/charge

        [HttpPost("charge")]

        public async Task<IActionResult> ProcessPayment(int orderId)

        {

            var order = await _context.Orders.Include(o =>
o.OrderItems).FirstOrDefaultAsync(o => o.Id == orderId);


            if (order == null)

            {

                return NotFound("Order not found.");

            }
            if (order.PaymentStatus == "Paid")

            {

                  return BadRequest("Order is already paid.");

            }


            try

            {

                  // Create a payment intent

                  var paymentIntentService = new PaymentIntentService();

                  var paymentIntent = paymentIntentService.Create(new
PaymentIntentCreateOptions

                  {

                  Amount = (long)(order.TotalAmount * 100), // Convert
to cents

                  Currency = "usd",

                  PaymentMethodTypes = new List<string> { "card" },

                  });


                  // Save the payment status as successful

                  order.PaymentStatus = "Paid";

                  _context.Entry(order).State = EntityState.Modified;

                  await _context.SaveChangesAsync();
```

```
                return Ok(new

                {

                Message = "Payment processed successfully.",

                PaymentIntentId = paymentIntent.Id

                });

            }

            catch (StripeException ex)

            {

                return BadRequest(new { Error = ex.Message });

            }

        }

        }

    }
```

6. Test the Payment Endpoint

Use Postman or another API tool to test payment processing:

1. **Process payment for an order**:
    - Endpoint: `POST /api/Payments/charge?orderId=1`
    - Response:

```
{

  "message": "Payment processed successfully.",

  "paymentIntentId": "pi_1234567890"

}
```

7. Update the Orders Controller

Include the PaymentStatus property when returning orders.

```
[HttpGet("{id}")]

public async Task<ActionResult<Order>> GetOrder(int id)

{

      var order = await _context.Orders

      .Include(o => o.Customer)

      .Include(o => o.OrderItems)

      .ThenInclude(oi => oi.Product)

      .FirstOrDefaultAsync(o => o.Id == id);


      if (order == null)

      {

      return NotFound();

      }


      return Ok(order);

}
```

8. Enhance Payment Logic

- **Refunds**: Add a refund endpoint to reverse payments.

- **Payment Status Tracking**: Use webhooks to update the payment status in real-time.
- **Multiple Payment Methods**: Extend support for other payment methods like PayPal, Apple Pay, or Google Pay.

9. Test End-to-End Payment Flow

1. **Create an order**.
2. **Process payment** for the order using the `PaymentsController`.
3. Verify the `PaymentStatus` is updated in the database.

10. (Optional) Add Webhook Support

To handle Stripe webhooks for real-time payment updates:

- Create an endpoint to process Stripe webhook events.
- Update the order `PaymentStatus` based on event types (e.g., `payment_intent.succeeded`).

Part 10: Logging and Error Handling

In this part, we'll implement logging and centralized error handling for the e-commerce API. This ensures errors are properly captured, logged, and presented to the client in a consistent format.

1. Install Logging Packages

Install Serilog for logging:

```
dotnet add package Serilog.AspNetCore

dotnet add package Serilog.Sinks.Console

dotnet add package Serilog.Sinks.File
```

2. Configure Serilog

- Update `Program.cs` to configure Serilog:

```
using Serilog;
```

```
}


var builder = WebApplication.CreateBuilder(args);


// Add Serilog configuration

Log.Logger = new LoggerConfiguration()

      .WriteTo.Console()

      .WriteTo.File("logs/log-.txt", rollingInterval:
RollingInterval.Day)

      .CreateLogger();


builder.Host.UseSerilog();


builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();


var app = builder.Build();


if (app.Environment.IsDevelopment())

{

      app.UseSwagger();

      app.UseSwaggerUI();

}
```

```
app.UseSerilogRequestLogging(); // Log HTTP requests


app.UseHttpsRedirection();

app.UseAuthentication();

app.UseAuthorization();


app.MapControllers();


app.Run();
```

The `Serilog.Sinks.File` ensures logs are written to a file in the `logs` directory.

3. Create a Global Error Handling Middleware

Add a middleware to handle exceptions globally.

- Create a new class `Middlewares/ErrorHandlerMiddleware.cs`:

```
using Microsoft.AspNetCore.Http;

using Serilog;

using System.Net;

using System.Text.Json;


namespace ECommerceAPI.Middlewares
```

```
{
      public class ErrorHandlerMiddleware
      {
      private readonly RequestDelegate _next;

      public ErrorHandlerMiddleware(RequestDelegate next)
      {
            _next = next;
      }

      public async Task Invoke(HttpContext context)
      {
            try
            {
                  await _next(context);
            }
            catch (Exception ex)
            {
                  await HandleExceptionAsync(context, ex);
            }
      }
      private static Task HandleExceptionAsync(HttpContext context,
Exception ex)

      {

            Log.Error(ex, "An error occurred while processing the
request.");


            var response = context.Response;

            response.ContentType = "application/json";

            response.StatusCode =
(int)HttpStatusCode.InternalServerError;


            var result = JsonSerializer.Serialize(new

            {

                  Message = "An unexpected error occurred.",

                  Details = ex.Message

            });


            return response.WriteAsync(result);

      }

      }

}
```

Register the middleware in Program.cs:

```
app.UseMiddleware<ECommerceAPI.Middlewares.ErrorHandlerMiddleware>();
```

4. Use Logging in Controllers

Enhance controllers to log important events.

Example: Update the ProductsController:

```
[HttpPost]

public async Task<ActionResult<Product>> PostProduct(Product product)

{

      try

      {

      _context.Products.Add(product);

      await _context.SaveChangesAsync();


      Log.Information("Product {ProductName} created with ID
{ProductId}", product.Name, product.Id);


      return CreatedAtAction(nameof(GetProduct), new { id = product.Id
}, product);

      }

      catch (Exception ex)

      {

      Log.Error(ex, "Failed to create product {ProductName}",
product.Name);

      throw; // Let the error middleware handle the exception

      }
}
```

5. Add Detailed Error Responses

Modify responses to include meaningful error messages.

Example: Validate product creation:

```
if (string.IsNullOrEmpty(product.Name))
{
    Log.Warning("Product creation failed due to missing name.");
    return BadRequest("Product name is required.");
}
```

6. Test the Error Handling

1. **Trigger an error**:
   - Call an endpoint with invalid data (e.g., `POST /api/Products` with an empty body).
   - Observe the structured error response and logs:
     - Console output.
     - Log file in the `logs` directory.
2. **Simulate an exception**:
   - Temporarily throw an exception in a controller:

```
throw new Exception("Test exception.");
```

Observe the global error handling behavior.

7. Future Enhancements

- **External Logging Services**: Use external logging platforms like Seq, Elasticsearch, or Azure Application Insights for advanced analytics and monitoring.
- **Custom Error Codes**: Add custom error codes to help clients handle errors programmatically.

Example:

```
{
  "Message": "Invalid product data.",
  "ErrorCode": "ERR_INVALID_PRODUCT"
}
```

Part 11: API Versioning and Documentation

In this part, we'll implement API versioning to support future updates without breaking existing clients. Additionally, we'll enhance the documentation using Swagger/OpenAPI to provide detailed information about the API.

---

1. Install API Versioning Package

Install the required NuGet package for API versioning:

```
dotnet add package Microsoft.AspNetCore.Mvc.Versioning
```

2. Configure API Versioning

- Add API versioning in Program.cs:

```
builder.Services.AddApiVersioning(options =>
{
      options.DefaultApiVersion = new ApiVersion(1, 0); // Default to
version 1.0
      options.AssumeDefaultVersionWhenUnspecified = true;
      options.ReportApiVersions = true; // Include version information
in the response headers
});
```

3. Version the Controllers

- Update an existing controller (e.g., `ProductsController`) to specify its API version:

```
using Microsoft.AspNetCore.Mvc;


[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiController]
public class ProductsController : ControllerBase
{
      // Existing code
}
```

Add another version for demonstration purposes:

- Duplicate `ProductsController` and name it `ProductsV2Controller`:
  - Change the API version to `2.0`.
  - Update the route:

```
[ApiVersion("2.0")]

[Route("api/v{version:apiVersion}/[controller]")]

[ApiController]

public class ProductsV2Controller : ControllerBase

{

     // Modify the controller to introduce changes for version 2.0

     [HttpGet]

     public ActionResult<string> GetProducts()

     {

     return "This is version 2.0 of the Products API.";

     }

}
```

4. Test API Versioning

1. **Access version 1.0**:
   - `GET /api/v1/Products`
2. **Access version 2.0**:
   - `GET /api/v2/Products`
3. **Check versioning headers**:
   - Observe the `api-supported-versions` and `api-deprecated-versions` headers in the response.

5. Install Swagger/OpenAPI for Documentation

Install the required Swagger package:

```
dotnet add package Swashbuckle.AspNetCore
```

6. Configure Swagger

- Add Swagger services in Program.cs:

```
builder.Services.AddSwaggerGen(options =>
{
     options.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo
     {
     Version = "1.0",
     Title = "E-Commerce API",
     Description = "API for managing an e-commerce platform (v1.0)"
     });

     options.SwaggerDoc("v2", new Microsoft.OpenApi.Models.OpenApiInfo
     {
     Version = "2.0",
     Title = "E-Commerce API",
     Description = "API for managing an e-commerce platform (v2.0)"
     });
});
```

Enable Swagger middleware in Program.cs:

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();

    app.UseSwaggerUI(options =>

    {

    options.SwaggerEndpoint("/swagger/v1/swagger.json", "E-Commerce
API v1.0");

    options.SwaggerEndpoint("/swagger/v2/swagger.json", "E-Commerce
API v2.0");

    });

}
```

7. Add Annotations to Enhance Documentation

- Use attributes to improve the Swagger documentation for your endpoints.

Example: Update ProductsController methods:

```
using Microsoft.AspNetCore.Mvc;

using System.Collections.Generic;


[ApiVersion("1.0")]

[Route("api/v{version:apiVersion}/[controller]")]

[ApiController]

public class ProductsController : ControllerBase

{

    /// <summary>

    /// Retrieves all products.
    /// </summary>

    /// <returns>A list of products.</returns>

    [HttpGet]

    public ActionResult<IEnumerable<string>> GetProducts()

    {

    return new[] { "Product1", "Product2" };

    }


    /// <summary>

    /// Retrieves a specific product by ID.

    /// </summary>

    /// <param name="id">The product ID.</param>

    /// <returns>The requested product.</returns>

    [HttpGet("{id}")]

    public ActionResult<string> GetProduct(int id)

    {

    return $"Product {id}";

    }

}
```

8. Test the Documentation

1. Run the application and navigate to the Swagger UI:
   - Example: http://localhost:5000/swagger

2. Select the API version (e.g., v1.0 or v2.0) and test the endpoints directly from the Swagger UI.

---

9. Future Enhancements

- **Deprecation Notices**: Mark older versions as deprecated.
- **Authentication Support**: Add JWT authentication to Swagger for testing secured endpoints.
- **Custom Styling**: Customize the Swagger UI to match your branding.

Part 12: Deployment

In this part, we'll go through the process of deploying your e-commerce API to a cloud platform. We'll use Azure App Service as an example, but the steps can be adapted to other platforms like AWS, Google Cloud, or Heroku.

1. Prerequisites

1. **Azure Account**: Create a free or paid account at Azure.
2. **Azure CLI**: Install the Azure CLI tool: Installation Guide.
3. **SQL Server**: If using Azure SQL Database, ensure you have a database instance created.

2. Prepare the Application

- Ensure your application is production-ready:
    1. Set the `ASPNETCORE_ENVIRONMENT` to `Production`.
    2. Update `appsettings.json` with the production database connection string.

Example:

```
"ConnectionStrings": {

  "DefaultConnection":
"Server=<azure-sql-server>.database.windows.net;Database=<your-databas
e>;User Id=<username>;Password=<password>;"

}
```

3. Publish the Application

1. Open a terminal in your project directory.
2. Publish the application using the following command:

```
dotnet publish -c Release -o ./publish
```

This creates a publish folder with all the files required for deployment.

4. Create an Azure App Service

1. **Log in to Azure**:

```
az login
```

Create a Resource Group:

```
az group create --name ECommerceResourceGroup --location "East US"
```

Create an App Service Plan:

```
az appservice plan create --name ECommercePlan --resource-group
ECommerceResourceGroup --sku B1 --is-linux
```

Create a Web App:

```
az webapp create --resource-group ECommerceResourceGroup --plan
ECommercePlan --name ECommerceAPI --runtime "DOTNET|8.0"
```

5. Deploy the Application

1. **Deploy Files to Azure**:

```
az webapp deploy --resource-group ECommerceResourceGroup --name
ECommerceAPI --src-path ./publish
```

Set Environment Variables: Add environment variables (e.g., connection strings) to the web app:

```
az webapp config appsettings set --resource-group
ECommerceResourceGroup --name ECommerceAPI --settings
ASPNETCORE_ENVIRONMENT=Production
```

Verify Deployment: Open the app in your browser using the URL provided by Azure:

```
https://ECommerceAPI.azurewebsites.net
```

6. Set Up Azure SQL Database

1. Create an Azure SQL Database instance:

```
az sql server create --name ECommerceSQLServer --resource-group
ECommerceResourceGroup --location "East US" --admin-user <username>
--admin-password <password>
```

```
az sql db create --resource-group ECommerceResourceGroup --server
ECommerceSQLServer --name ECommerceDB --service-objective S0
```

Configure the database connection in `appsettings.json` (already done in step 2).

Migrate the database:

```
dotnet ef database update
```

7. Configure Logging

Enable application logging in Azure to monitor your app.

```
az webapp log config --resource-group ECommerceResourceGroup --name
ECommerceAPI --application-logging true --level information
```

View logs:

```
az webapp log tail --resource-group ECommerceResourceGroup --name
ECommerceAPI
```

8. Test the Deployed Application

1. Test endpoints like `GET /api/Products` or `POST /api/Orders` using the live API URL.

2. Check the Swagger documentation if enabled: https://ECommerceAPI.azurewebsites.net/swagger.

9. Future Enhancements

- **Custom Domain**: Map your web app to a custom domain.
- **SSL Certificates**: Enable HTTPS for secure communication.
- **Scaling**: Upgrade the App Service Plan for higher performance or enable auto-scaling.