## Web API

### Exercise 1: Develop a WEB API Project for Product Management

Develop a WEB API project for product management in .NET Core. Follow the steps below:

* Develop a new ASP.NET Core Web API project
* Implement the necessary models, controllers, and actions for product management. The **Product** model should include properties like

public class Product

{

public int Id { get; set; }

public string Name { get; set; }

public decimal Price { get; set; }

public string Description { get; set; }

}

Use static data (e.g., a list of products) to simulate a database. Initialize the list of products in memory in the controller. For this you create class **ProductService**

* Define a class **ProductService** representing a service for managing products.
* It should contain a private list products **List<Product>** to store product data and initializes it with static product instances.
* The class provides methods to perform CRUD (Create, Read, Update, Delete) operations on products, such as retrieving all products, getting a product by its ID, adding a new product, updating the product and deleting the product.
* The service utilizes in-memory storage and employs LINQ to query and manipulate the product data.

Sample **List<product>**

new Product { Id = 1, Name = "Product 1", Price = 10.99m, Description = "Description 1" },

new Product { Id = 2, Name = "Product 2", Price = 20.49m, Description = "Description 2" },

new Product { Id = 3, Name = "Product 3", Price = 15.99m, Description = "Description 3" }

Create **ProductController** to implement HTTP methods for managing products:

**GetAllProducts()** **HTTP GET -** Get Method:

Retrieves a list of all product.

**Return Type**: **IActionResult**

Make a GET request to **/api/product**

**GetProductById(int productId) HTTP GET with Parameter -** Get Method:

Retrieves details of a specific product by its Id.

**Return Type**: **IActionResult**

Make a GET request to **api/product/{productId}** where {productId} is the Id of the product.

**CreateProduct([FromBody] Product newProduct) HTTP POST - Post Method:**

Use the HTTP POST method to create a new product.

Make a POST request to **/api/product** with the product data in the request body.

**Return Type**: **IActionResult**

If the creation is successful, the server returns a **201** Created response with the newly created product data.

**UpdateProduct(int id, [FromBody] Product updatedProduct) HTTP PUT -** Put Method:

Updates an existing product with the provided data.

Make a PUT request to **/api/product/{id}** with the product data in the request body.

**Return Type**: **IActionResult**

If the product with the specified id does not exist, the server returns a 404 Not Found.

If the update is successful, the server returns a 200 OK with the updated product data.

**DeleteProduct(int productId) HTTP DELETE -** Delete Method:

Deletes an existing product by its ID.

Make a DELETE request to **/api/product/{id}** with the product Id in the request body.

**Return Type**: **IActionResult**

If the product with the specified id does not exist, the server returns a 404 Not Found.

If the deletion is successful, the server returns a 204 No Content.

Create **ProductService** to implement HTTP methods for managing products:

**GetAllProducts()** - Retrieves a list of all products. The return type must be **List.**

**GetProductById(int productId) -** Retrieves details of a specific product by its Id. The return type must be **Product**. This method returns a single product that matches the provided productId from the list of products.

**AddProduct(Product newProduct) -** Add a new product. The return type must be **void**.

**UpdateProduct(int id,** **Product updatedProduct)** - Updates the specific product details. The return type must be **void**.

**DeleteProduct(int id)** - Deletes a specific product details. The return type must be **void**.

Ensure that the project follows coding standards and guidelines. Test the API endpoints using Swagger to verify their functionality.

**Run the application with above mentioned API endpoints and click Run Testcase Button.**

**Implement Different Status Codes**

Implement different HTTP status codes for the API responses. Follow the guidelines below:

1.     For successful operations (20X):

* Use **200 OK** when returning a list of product or retrieving a specific product.
* Use **201 Created** when a new product is successfully created using the HTTP POST method.
* Use **CreatedAtAction** or **CreatedAtRoute** to provide the URL for the newly created resource in the response header.

2.     For client errors (40X):

* Use **400 BadRequest** when the request payload is invalid or missing required data.
* Use **404 NotFound** when an product is not found by its ID during a GET operation.

Ensure that the API returns appropriate status codes based on the operation results and follows coding standards and guidelines.

Implement Different HTTP Status Codes:

a. Update the Get method:

**Return NoContent (HTTP 204)** when no products are found.

b. Update the Get method with a parameter:

**Return NotFound (HTTP 404)** when the products is not found.

c. Update the Post method:

**Return BadRequest (HTTP 400)** when the request payload is invalid or missing.

**Return CreatedAtAction or CreatedAtRoute (HTTP 201)** to provide the URL for the newly created resource in the response header.

**Implement Swagger documentation for the Product Management WEB API project.**

### Exercise 2: Develop a WEB API Project for Order Management

Develop a WEB API project for order management in .NET Core. Follow the steps below:

Develop a new ASP.NET Core Web API project.

* Implement the necessary models, controllers, and actions for order management.
* The Order model should include properties like:

public class Order

{

public int OrderId { get; set; }

public string CustomerName { get; set; }

public DateTime OrderDate { get; set; }

public decimal TotalAmount { get; set; }

public string Status { get; set; }

}

Use static data (e.g., a list of orders) to simulate a database. Initialize the list of orders in memory in the controller. For this, create a class OrderService.

* Define a class **OrderService** representing a service for managing orders.
* It should contain a private list **List<Order>** to store order data and initializes it with static order instances.
* The class provides methods to perform CRUD (Create, Read, Update, Delete) operations on orders, such as retrieving all orders, getting an order by its ID, adding a new order, updating an existing order, and deleting an order.
* The service utilizes in-memory storage and employs LINQ to query and manipulate the order data.

Sample **List<Order>**

new Order { OrderId = 1, CustomerName = "John Doe", OrderDate = new DateTime(2023, 1, 1), TotalAmount = 100.50m, Status = "Shipped" },

new Order { OrderId = 2, CustomerName = "Jane Smith", OrderDate = new DateTime(2023, 2, 15), TotalAmount = 250.75m, Status = "Processing" },

new Order { OrderId = 3, CustomerName = "Alice Johnson", OrderDate = new DateTime(2023, 3, 20), TotalAmount = 150.00m, Status = "Delivered" }

Create **OrderController** to implement HTTP methods for managing orders: (namespace dotnetapp.Controllers)

**GetAllOrders()** **HTTP GET -** Get Method:

Retrieves a list of all order.

Make a GET request to **/api/Order**

**GetOrderById(int orderId) HTTP GET with Parameter -** Get Method:

Retrieves details of a specific order by its Id.

Make a GET request to **api/Order/{orderId}** where {**orderId**} is the Id of the order.

**CreateOrder(Order newOrder)** **HTTP POST -** Post Method**:**

Use the HTTP POST method to create a new order.

Make a POST request to **/api/Order** with the order data in the request body.

Create **OrderService** to implement HTTP methods for managing orders: (namespace dotnetapp.Services)

**GetAllOrders()** - Retrieves a list of all orders.

**GetOrderById(int orderId) -** Retrieves details of a specific order by its Id.

**AddOrder(Order newOrder) -** Add a new order.

Ensure that the project follows coding standards and guidelines. Test the API endpoints using Swagger to verify their functionality.

**Run the application with above mentioned API endpoints and click Run Testcase Button.**

**Implement Different Status Codes**

Implement different HTTP status codes for the API responses. Follow the guidelines below:

1.     For successful operations (20X):

* Use **200 OK** when returning a list of order or retrieving a specific order.
* Use **201 Created** when a new order is successfully created using the HTTP POST method.
* Use **CreatedAtAction** or **CreatedAtRoute** to provide the URL for the newly created resource in the response header.

2.     For client errors (40X):

* Use **400 BadRequest** when the request payload is invalid or missing required data.
* Use **404 NotFound** when an order is not found by its ID during a GET operation.

Ensure that the API returns appropriate status codes based on the operation results and follows coding standards and guidelines.

Implement Different HTTP Status Codes:

a. Update the Get method:

**Return OK (HTTP 200)** when returning a list of orders.

b. Update the Get method with a parameter:

**Return OK (HTTP 200)** when returning a list of orders with respective id.

**Return NotFound (HTTP 404)** when the order is not found.

c. Update the Post method:

**Return BadRequest (HTTP 400)** when the request payload is invalid or missing.

**Return CreatedAtAction or CreatedAtRoute (HTTP 201)** to provide the URL for the newly created resource in the response header.

**Implement Swagger documentation for the Order Management WEB API project.**