Order API Documentation

Preface

This document is intended to describe how to properly use and consume the API developed for this technical test with HTTP requests.

Order API

**Base URL**

The Base URL is the root URL for all of the API. The base URL for the API is: <http://localhost:5656/api/>

**Orders Controller**

The orders controller holds all of the operations regarding orders. The base URL for the orders controller is: <http://localhost:5656/api/orders>

The following operations are supported:

Create an order

Http request type: POST

Url: <http://localhost:5656/api/orders>

Descritpion: Creates an order. If there are no errors, returns the order create as a Json object

Request example:

{

"CustomerId": "10",

"DeliveryAddress": "Dublin St 1",

"ProductId": "1",

"Quantity": "2",

"UnitPrice": "5"

}

Response example:

{

"id": 1,

"customerId": 10,

"productId": 1,

"quantity": 2,

"unitPrice": 5,

"deliveryAddress": "Dublin St 1",

"orderStatus": 1

}

List all orders

Http request type: GET

Url: <http://localhost:5656/api/orders>

Description: Get all orders found in the database

Request example: <http://localhost:5656/api/orders>

Response example:

[

{

"orderId": 15,

"customerId": 10,

"productId": 25,

"deliveryAddress": "Dummy Street 1",

"quantity": 2,

"unitPrice": 10,

"orderStatus": 2,

"orderStatusDescription": "Completed",

"orderTotal": 20

},

{

"orderId": 16,

"customerId": 22482,

"productId": 1,

"deliveryAddress": "Dummy Address",

"quantity": 2,

"unitPrice": 10,

"orderStatus": 1,

"orderStatusDescription": "Pending",

"orderTotal": 20

}

]

Get by order Id

Http request type: GET

Url: <http://localhost:5656/api/orders>/{id:int}

Description: Gets an order by its ID. If none is found, returns a message stating that the order with {id} was not found

Request example: <http://localhost:5656/api/orders>/15

Response example:

{

"orderId": 15,

"customerId": 10,

"productId": 25,

"deliveryAddress": "Dummy Street 1",

"quantity": 2,

"unitPrice": 10,

"orderStatus": 2,

"orderStatusDescription": "Completed",

"orderTotal": 20

}

Query orders

Http request type: POST

Url: <http://localhost:5656/api/orders>/query

Description: Searches for orders within a given criteria.

Supports the following attributes:

OrderId - int

CustomerId - int

ProductId - int

DeliveryAddress - int

Request example:

{

"DeliveryAddress": "Dummy"

}

Response example:

[

{

"orderId": 15,

"customerId": 10,

"productId": 25,

"deliveryAddress": "Dummy Street 1",

"quantity": 2,

"unitPrice": 10,

"orderStatus": 2,

"orderStatusDescription": "Completed",

"orderTotal": 20

},

{

"orderId": 16,

"customerId": 22482,

"productId": 1,

"deliveryAddress": "Dummy Address",

"quantity": 2,

"unitPrice": 10,

"orderStatus": 1,

"orderStatusDescription": "Pending",

"orderTotal": 20

}

]

Thoughts and considerations

The following text is not an actual part of the document, but more like what I have to say about the solution. Some of the sentences may have relevant information for the readers.

It is clear that I have never done any kind of documentation before, so I tried to write something that was brief and made some sense regarding the API.

I did not track the amount of time that I took to finish the whole test (excluding the documentation), but I can give a rough estimate of 6 to 8 hours at most, with almost half of these hours being spent on writing tests for the application and refactoring.

I decided against using some frameworks for the project for simplicity’s sake.

* Dependency Injection is made through controllers using the native Asp.Net Core DI framework
* Mapping is done manually through a static class
* Database mapping and management is done automatically with EF Core
* MS Test was used as the testing framework of choice
* Mocking framework (such as Moq) were not used in unit tests in favour of using In Memory database with EF Core. The reasoning being that there was not enough stuff to be worth faking with Moq

Customer and Product entities were omitted because I felt they were outside of the test’s scope. I believe that if I had implemented these entities I would have taken more time to finish the project and all I would prove is that I know how to use EF Core to create navigation fields in its entities.

The Order entity has Id properties that act as foreign keys for the aforementioned entities.

Acceptance tests were created to test the API behaviour. The Api endpoint which the tests connect to is hardcoded in the base acceptance test class. Ideally, the endpoint should be read from a configuration file, but, again, I felt this was outside of the test scope, so I chose to hardcode it into a class.

Validations for the Order entity were done through a chain of if statements in the Service class. It would be better if these validations were somewhere else, or in a ValidationRule object for the Order entity, but I chose to leave that out for the time being.

I wanted to ask some questions about the product requirements described in the tech test document, mainly the requirement number 2, points A and B (regarding when an Order should be rejected). Since I did not have any kind of way to ask these things, I interpreted these points the way I thought it made more sense. In a day to day work situation, I would have looked for someone who could answer me the doubts I had at the time.

Overall, I found the test OK and not too challenging. I tried to stick with the main goals of the test and keep things as simple as possible.

I do have another project that I did while interviewing for another company a month ago. The project goal was to consume an API and do some kind of calculus using its results to present a specific set of data. It is a simple console application in .Net with Unit tests and Acceptance tests. If time is not an issue, it is worth taking a look at it due to its higher level of complexity: <https://github.com/TonySniper/StarWars.StarShips>