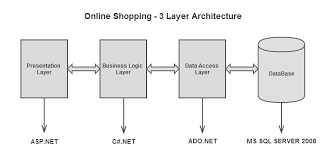
# Architecture – ONLINE STORE

This step is one of the essential steps before we begin the [**REST API**](https://novicedeveloper.com/what-is-rest-api/) development. Without clearly defining the architecture of the system we cannot build it without any issue.

The below image represents the system that we are building. Our system consists of a RESTful service and a database. The RESTful service, **Product Service**, will be developed in Node.js with Express.js. MS SQL SERVER will be used as a database for the service to interact.



Architecture – ONLINE SHOPPING STORE

# Low-Level Design

So far we have defined the architecture for the customer information system. Now it’s the time to do a deep dive into the low-level design of the API. This step is the next one after the architecture.

In the low-level design, we will go to the API endpoint level design. After the request has been received by the RESTful service what will happen to service that request is what we will discuss in the low-level design of each of the API endpoints.

Usually, we will have to create the sequence diagram for each endpoint at this stage.

Before we jump onto the sequence diagrams it’s better to look at what are all the different components will be involved right from the request received stage until the response is sent back to the requester.

# Layers Or Components Of Node.js REST API

Typically, RESTful service will accomplish the task of serving the request in the layered approach. In the Node.js world, there are different components available for us to implement the API functionality. They are route, middleware, service, and model.

Some of them may not be [**Node.js**](https://nodejs.org/) framework related layers, but they not new to the restful service world. In object-oriented languages, the services are most probably implemented in such a layered approach. We could borrow that approach here as well.

Below are the different components that we will be using to implement the **Product Service**.

Let’s have a brief look at these components and see what each will do.

## 1. Controller

Usually, a controller will handle the request, invoke services to perform that action, and process response to sending back to the requester. Often controller will make a sequence of service calls in orchestration to accomplish the request as designed. Technically, it handles the flow of the middleware calls before it sends the response.

## 2. Middleware

Middleware in a NodeJS world is a function that has access to the request object, response object, and next function. Here the **next()** function is used to invoke the next middleware in the stack.

Middleware functions can perform the following tasks:

1. Execute any code.
2. Make changes to the request and the response objects.
3. End the request-response cycle.
4. Call the next middleware in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

## 3. Service

A service is any function that can perform any task, like calculating some formula, accessing the database to read or write. Here we will use a service function to access the database for retrieving and storing the customer information.

Also, the service will not have access to the request and response object. So anything needs to be done on the request and response object will have to be done in the controller only.

Then, the controller will have to pass that information as parameters to the service functions to perform the task. This way there is a clear segregation of duties among all the components in the RESTful service.

## 4. Model

This component is the data access layer to fetch and save the documents. The service layer will be invoking the models to perform any actions on the document in the database via the model.

A model represents the document that can be created, updated, removed and fetched from the database.

# REST API Sequence Diagrams

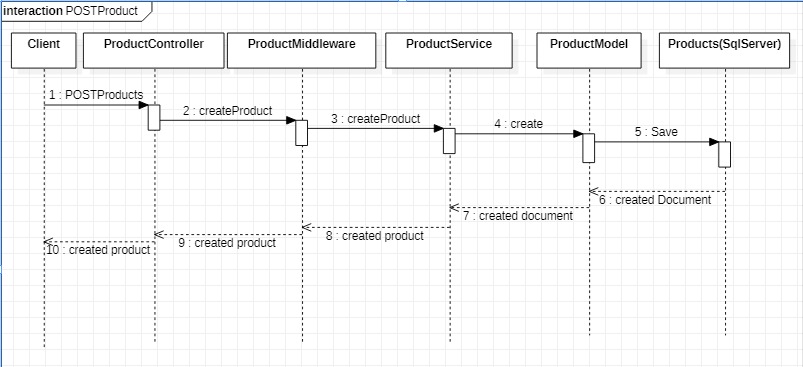
In the above sections, we have gone through all the components/layers we will be engaging to implement the RESTful service. The next step would be to see how can we use these layers to perform each of the functionalities that we defined in the requirements section.

A sequence diagram would be a perfect tool to visualize and describe the flow of the process to develop the code. In this section, we will go through the sequence diagrams for each functionality.

This diagram will be very helpful during the development process as it sets the goal of what we are going to develop for those functionalities. With this clear end goal laid out, we can quickly and easily continue with the development and testing of the robust API.

## 1. Add Product API – Sequence Diagram

The below image shows the sequence diagram for the add new product functionality. As you can see, the request comes to the controller; then it goes through the middleware; finally, it’s handled by the model to create the document in the database.

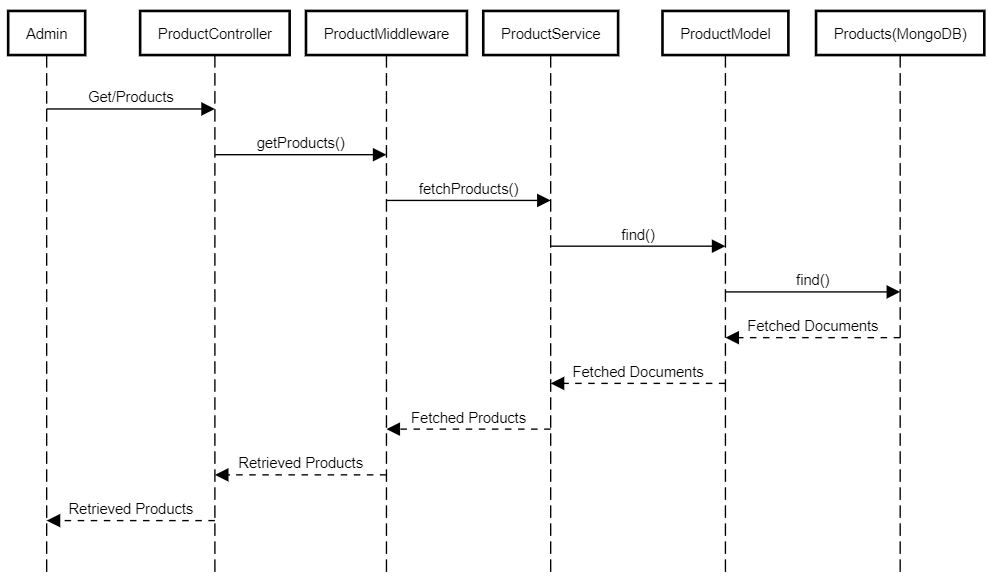


**Add Product API – Sequence Diagram**

## 2. Get Product List API – Sequence Diagram

The HTTP GET request will be sent to the controller to get the product’s list. The request query parameters will be parsed to extract any searching, filtering and paging information within the controller.

Then the list of products will be retrieved for the search, filter and paging query from the database via the service and model. Finally, the retrieved product list will be sent back to the requester in the response object.

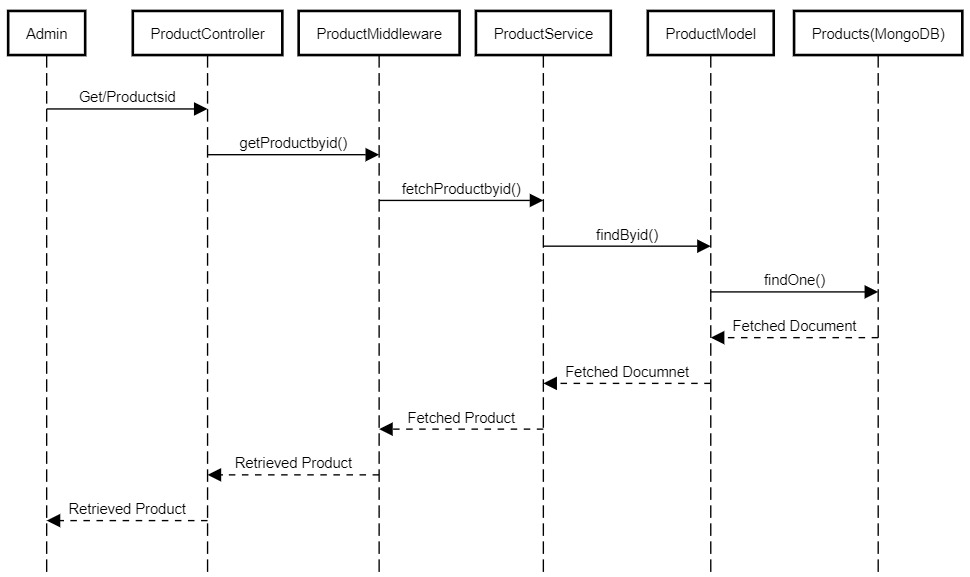


**Get Product List API – Sequence Diagram**

## 3. Get Product Details API – Sequence Diagram

What we have here is the sequence diagram for the get a customer functionality. This flow starts with the request comes with the product id to retrieve the information.

As in other sequences, the controller receives the request and retrieves the product details through the service and model from the database.

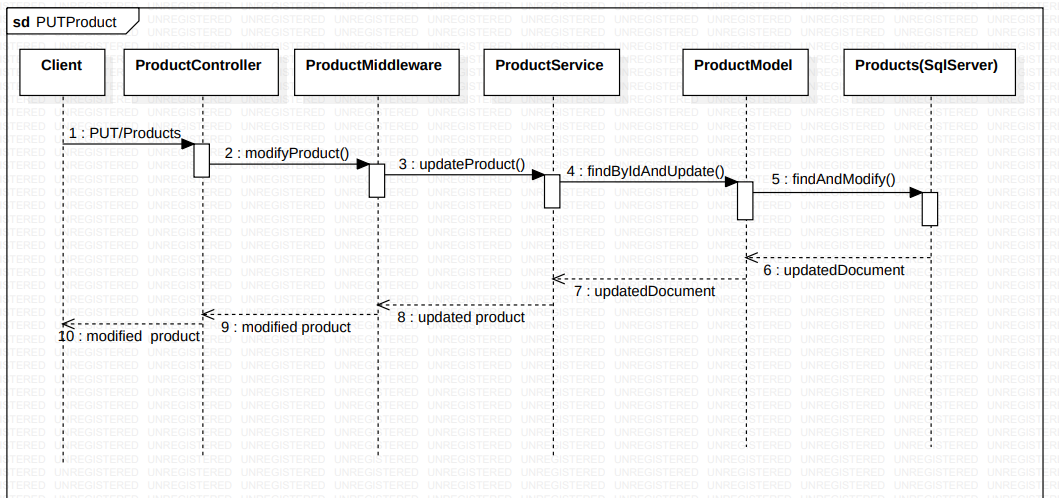


**Get Product Details API – Sequence Diagram**

## 4. Modify Product API – Sequence Diagram

The sequence diagram for modifying product functionality is shown here. As per this flow, the request comes with the changed customer details, and the controller handles it.

Before the controller responds with the modification result, it invokes the *modifyproduct* function in the service. In turn, the model is getting called from the service to make the document update in the database.

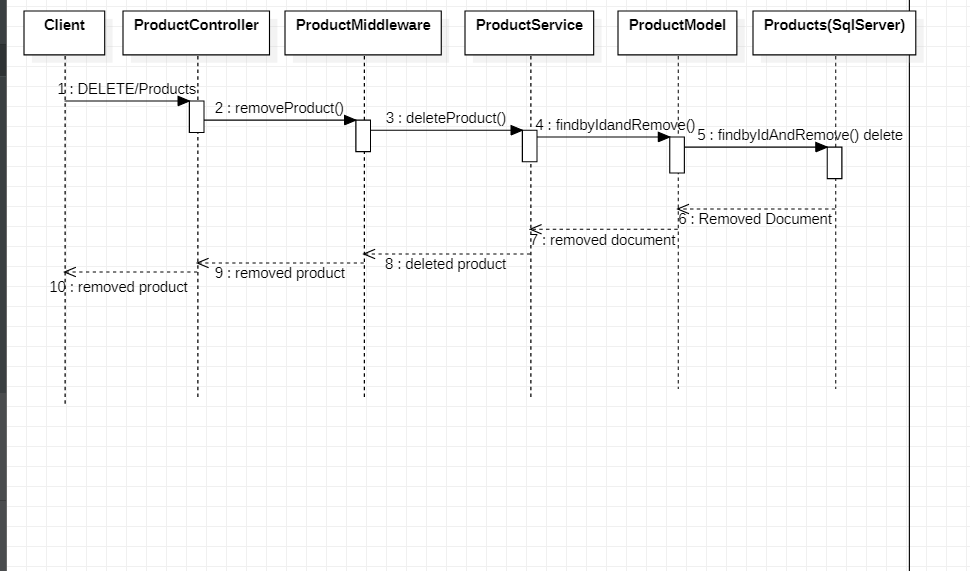


**Modify Product API - Sequence Diagram**

## 5. Delete Product API – Sequence Diagram

Just like the modify request, the delete a product request comes with the customer id which needs to be removed from the database.

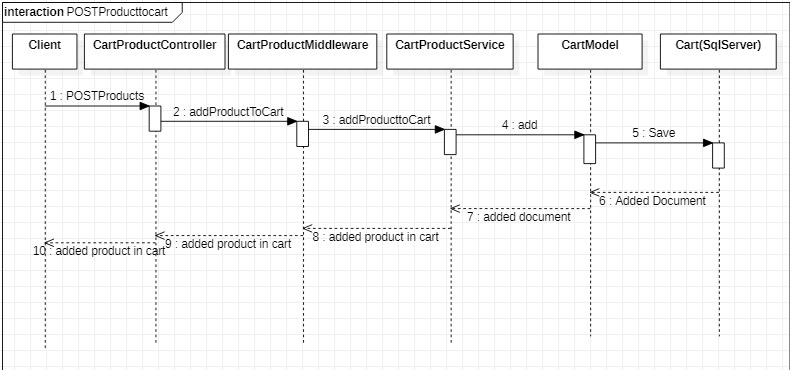
As usual, the controller initiates the process to delete the product from the database with the help of service and model components.



**Delete Product API – Sequence Diagram**

## 6. Add Cart API – Sequence Diagram

The below image shows the sequence diagram for the add new Cart functionality. As you can see, the request comes to the controller; then it goes through the middleware; finally, it’s handled by the model to create the document in the database.

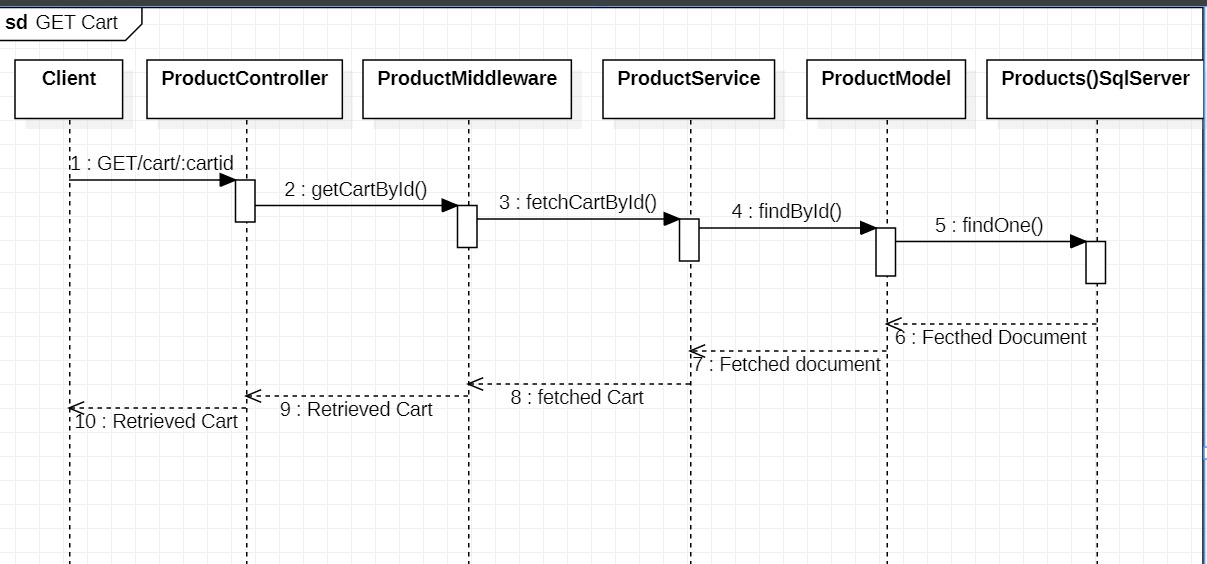


**Add Cart API – Sequence Diagram**

## 7. Get Cart API – Sequence Diagram

The HTTP GET request will be sent to the controller to get the Cart. The request query parameters will be parsed to extract any searching, filtering and paging information within the controller.

Then the list of products will be retrieved for the search, filter and paging query from the database via the service and model. Finally, the retrieved Cart list will be sent back to the requester in the response object.

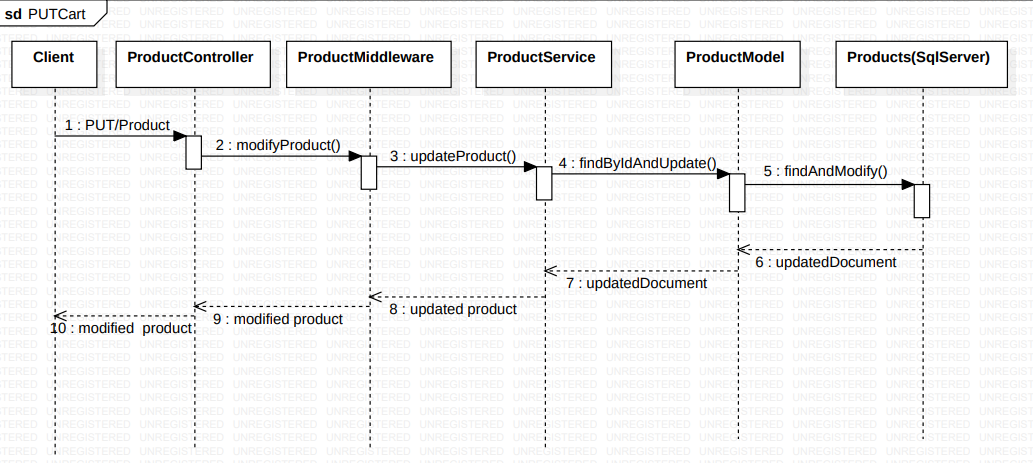


**Get Cart API – Sequence Diagram**

## 8. Modify Cart API – Sequence Diagram

The sequence diagram for modifying Cart functionality is shown here. As per this flow, the request comes with the changed Cart details, and the controller handles it.

Before the controller responds with the modification result, it invokes the *modifyCart* function in the service. In turn, the model is getting called from the service to make the document update in the database.

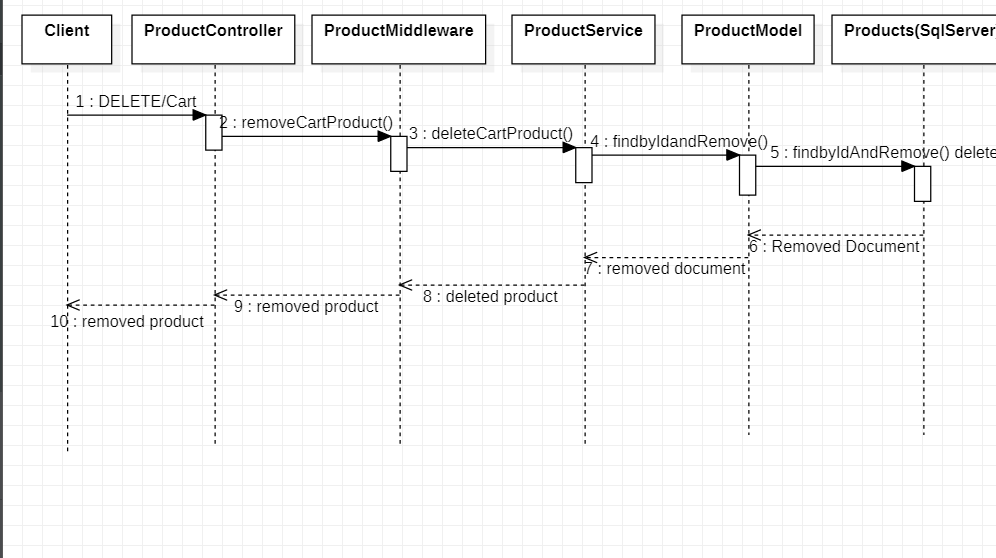


**Modify Cart API - Sequence Diagram**

## 9. Delete Cart API – Sequence Diagram

Just like the modify request, the delete a Cart request comes with the product id which needs to be removed from the database.

As usual, the controller initiates the process to delete the product from the Cart with the help of service and model components.



**Delete Cart API – Sequence Diagram**

**Mock APIs**

---

swagger: "2.0"

info:

description: "This is an API Which performs all the CRUD operations regarding the Online Store \n"

version: 1.0.0

title: Online Store

contact:

email: meghass029@gmail.com

license:

name: C# (dot NET)

url: http://www.apache.org/licenses/LICENSE-2.0.html

host: virtserver.swaggerhub.com

# basePath: /abcLimite/OnlineStore/1.0.0

tags:

- name: Products

description: Every details regarding Products

schemes:

- https

- http

paths:

/Products:

get:

tags:

- Products

summary: Details of Products

description: Get Product Details

operationId: getProductDetails

produces:

- application/json

parameters:

- name: productData

in: body

description: send a product data

schema:

$ref: '#/definitions/Products'

- name: productId

in: query

type: integer

format: int32

description: send a product Id

responses:

"200":

description: successful operation

schema:

type: array

items:

$ref: '#/definitions/Products'

"400":

description: Invalid status value

"401":

description: Unauthorized access

post:

tags:

- Products

summary: Adding the new Product

description: Adding a new Product

operationId: addProduct

consumes:

- application/json

produces:

- application/json

parameters:

- name: productData

in: body

description: send a product data

schema:

$ref: '#/definitions/Products'

responses:

"201":

description: Successfully created

"400":

description: Invalid status value

"401":

description: Unauthorized access

"409":

description: Duplicate record

put:

tags:

- Products

summary: updating the new Product

description: updating a new Product

operationId: updateProduct

consumes:

- application/json

produces:

- application/json

parameters:

- name: productData

in: body

description: send a product data

schema:

$ref: '#/definitions/Products'

- name: productId

in: query

type: integer

format: int32

description: send a product Id

responses:

"204":

description: Successfully updated

"400":

description: Invalid status value

"401":

description: Unauthorized access

"500":

description: Server error

delete:

tags:

- Products

summary: deleting the new Product

description: deleting a new Product

operationId: deleteProduct

produces:

- application/json

parameters:

- name: productId

in: query

type: integer

format: int32

description: send a product Id

responses:

"204":

description: Successfully deleted

"400":

description: Invalid status value

"401":

description: Unauthorized access

"500":

description: Server error

/ProductList:

get:

tags:

- ProductList

summary: Collection of the Products

description: Get all the Product list

operationId: getProductList

produces:

- application/json

parameters:

- name: limit

in: query

description: maximum number of records to return

required: false

type: integer

maximum: 500

minimum: 0

format: int32

responses:

"200":

description: successful operation

schema:

type: array

items:

$ref: '#/definitions/ProductList'

"400":

description: Invalid status value

"401":

description: Unauthorized access

/Cart:

post:

tags:

- Cart

summary: Adding a Product to cart

description: Adding a Product to cart

operationId: addProductToCart

consumes:

- application/json

produces:

- application/json

parameters:

- name: productData

in: body

description: send a product data

schema:

$ref: '#/definitions/Cart'

responses:

"201":

description: Successfully created

"400":

description: Invalid status value

"401":

description: Unauthorized access

"409":

description: Duplicate record

get:

tags:

- Cart

summary: Details of Cart

description: Get Cart Details

operationId: getCartDetails

produces:

- application/json

parameters:

- name: limit

in: query

description: maximum number of records to return

required: false

type: integer

maximum: 500

minimum: 0

format: int32

responses:

"200":

description: successful operation

schema:

type: array

items:

$ref: '#/definitions/Cart'

"400":

description: Invalid status value

"401":

description: Unauthorized access

delete:

tags:

- Cart

summary: deleting the cart Product

description: deleting a cart Product

operationId: deleteCartProduct

produces:

- application/json

parameters:

- name: productId

in: query

type: integer

format: int32

description: send a product Id

responses:

"204":

description: Successfully deleted

"400":

description: Invalid status value

"401":

description: Unauthorized access

"500":

description: Server error

put:

tags:

- Cart

summary: updating the product from cart

description: updating cart information

operationId: updateCart

consumes:

- application/json

produces:

- application/json

parameters:

- name: cartData

in: body

description: send a product data

schema:

$ref: '#/definitions/Cart'

- name: productId

in: query

type: integer

format: int32

description: Update quantity in cart

responses:

"204":

description: Successfully updated

"400":

description: Invalid status value

"401":

description: Unauthorized access

"500":

description: Server error

definitions:

Products:

type: object

required:

- productId

- productName

- price

- productDescription

- sellerName

- sellerContact

properties:

productId:

type: integer

format: int32

example: 101

productName:

type: string

example: Vivo mobile

price:

type: integer

format: int32

example: 20000

productDescription:

type: string

example: Android Phone

sellerName:

type: string

example: Sangeetha mobiles

sellerContact:

type: string

example: 987458963

Cart:

type: object

required:

- productId

- customerId

- shoppingId

- quantity

properties:

productId:

type: integer

format: int32

example: 101

customerId:

type: integer

format: int32

example: 1001

shoppingId:

type: integer

format: int32

example: 1

quantity:

type: integer

format: int32

example: 2

ProductList:

type: object

required:

- productId

- productName

- price

properties:

productId:

type: integer

format: int32

example: 101

productName:

type: string

example: Vivo mobile

price:

type: integer

format: int32

example: 20000

# Added by API Auto Mocking Plugin

basePath: /NewWave\_TeamB/onlinestore/1.0.0