**#MyRetail REST API**

MyRetail RESTful service provides the client application ability to:

1. Retrieve Product and Price information by Product Id

2. Send request to modify the price information in the database

**GET Product Information**

**Input**: The client application does a GET request at the path "/products/{id}" for a product

**Internal Working**: When the API receives the request, it sends a request to "redsky.target.com" and retrieves the product information. This product information doesn't contain price that is needed by the user. The price is retrieved from a data store. The price information is now combined with the required product information to provide only the required product information to the user.

**Output**: For a product with product id '13860428', the sample JSON output is as shown below

{"id":13860428,"name":"The Big Lebowski (Blu-ray) (Widescreen)","current\_price":{"value": 12.49,"currency\_code":"USD"}}

Errors/Validations: Appropriate error messages are provided after validating the data. The client application can use the message in the response to display the same to the user appropriately.

**Update Product Price in the datastore:**

Input: The user/client application can do a PUT request with input similar to the response received in GET and should be able to modify the price in the datastore. The request is done at the same path "/products/{id}"

**Sample Input**: JSON Body - {"id":13860428,"name":"The Big Lebowski (Blu-ray) (Widescreen)","current\_price":{"value": 15.67,"currency\_code":"USD"}}

**Internal Working**: When the API receives PUT request, it does some validations to see if the product id passed in URL and request body is same and also checks whether product is available in redsky service

If it is available, then the price for the product is modified in the data store and creates a ProductPriceinfo record if it is not available.

**Output**: Success message is returned if the price modification is done.

**Errors/Validations:** Appropriate error messages are provided after validating the data. The client application can use the message in the response to display the same to the user appropriately.

**Technology stack used:**

JDK 1.8) -http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Spring Boot - https://projects.spring.io/spring-boot/

MongoDB - https://www.mongodb.com/

Swagger - http://swagger.io/

Maven - https://maven.apache.org/

**Instructions to Setup**

Install JDK 1.8 (or 1.7) -http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Install MongoDB - https://docs.mongodb.com/manual/installation/

Install Maven - https://maven.apache.org/install.html

Run MongoDB - Run 'mongod.exe' in order to start Mongodb

Clone the code from git repository - <https://github.com/sgantala/myRetail.git>

Make sure you are in the MyRetail directory

Run the following command to start: mvn spring-boot:run

Open browser and visit Swagger. http://localhost:8888/swagger-ui.html

Swagger documentation explains the expected request and response for GET and PUT requests.

**Testing:**

The testcases are present in the folder 'src\test\java\com.myretail'.

All the test cases can be executed by running the following command from MyRetail root folder: mvn clean test

Test cases are available in src\test\java\com\myretail\service and src\test\java\com\myretail\controller

**Swagger UI:**

Swagger displays the following information for an API method by default.

Type of request(GET/PUT..) and the path of request

Status and format of the response

Response Content Type

Parameters list

Possible Failure Responses with HTTP code

The user can modify the values in the fields provided and can do "Try it out!" at the bottom. Please refer to the images **'Get\_Product\_Success.JPG'** and **'Get\_Product\_Success\_1.JPG**' to see what a sample GET and PUT requests look like.

More information about the API methods and the responses is provided below.

**API Requests and Responses**

**PUT Request:**

Following PUT request will store information of productID:13860428 in NOSQL database

**###Request:**

curl -X PUT --header "Content-Type: application/json" --header "Accept: application/json" -d "{

"productId": "13860428",

"title": "The Big Lebowski (Blu-ray)",

"currentPrice": {

"value": "113.33",

"currency\_code": "USD"

}

}" "http://localhost:8888/products/13860428"

**###Response:**

Status code: 201 { "response": "success" }

When productId in request url and body is different it will return 400 Bad Request

**###Request:**

curl -X PUT --header "Content-Type: application/json" --header "Accept: application/json" -d "{

"productId": "130428",

"title": "The Big Lebowski (Blu-ray)",

"currentPrice": {

"value": "113.33",

"currency\_code": "USD"

}

}" "http://localhost:8888/products/15117729"

**###Response:**

{

"timestamp": 1513570324175,

"status": 400,

"error": "Bad Request",

"exception": "com.myretail.exception.ProductMisMatchException",

"message": "Product id in request header and request body doesn't match",

"path": "/products/15117729"

}

**GET Request:**

Request:

curl -X GET --header 'Accept: application/json' 'http://localhost:8888/products/13860428'

Response:

{ "productId": "13860428", "title": "The Big Lebowski (Blu-ray)", "current\_price": { "value": "13.49", "currency\_code": "USD" } }

When you give a productID that doesn't exist you will get 404 Product not found.

Request:

curl -X GET --header 'Accept: application/json' 'http://localhost:8888/products/235345436'

Response:

{

"timestamp": 1513570517176,

"status": 404,

"error": "Not Found",

"exception": "com.myretail.exception.ProductNotFoundException",

"message": "Product not found",

"path": "/products/235345436"

}