# MyRetail Product Service Getting Started

Venkat Nimmaturi

# Table of Contents

# Introduction

myretail-product-service is designed as a microservice to retrieve product information. There are two pieces of information needed to be retrieved. Product Name is retrieved from another Restful service. The endpoint is made configurable so that if in case it changes in future, it can be modified in yml file easily.

The second piece of information that is retrieved is product price. Product price is stored in elasticsearch datastore in "products" index. Jest Client implementation is done to perform queries against elasticsearch as well as update price information in the Index.

The index information, elasticsearch connection parameters are made as configurable items. As the needs of MyRetail change, we have ability to configure accordingly with out having to make code changes and there by aavoiding deployment.

The application is designed as multiproject gradle repository. The idea behind this approach is to reuse the common libraries such as myretail-product-api, myretail-jest-common. Each subprojects are explained in subsequent pages.

# Application

The myretail-product-manager is ready to run out of the box. The setup for the standard server port is setup in application.yml

## Running inside of STS

when imported into STS you should see a standard runner on the "Boot Dashboard". Out of the box, you should only need Elasticsearch server running on http://localhost:9200

## Elasticsearch

Elasticsearch container is readily available in dockerhub and is available to run elasticsearch in virtual machine if setup. Once thing to make a note is you could setup portforwarding so that all the requests on localhost are forwarded to docker-machine’s IP

docker-compose is available at https://bitbucket.org/venkatnimmaturi/docker-main-ecosystem [Elasticsearch docker image]

## Consolidated instructions to run the project

- Install docker machine, docker-compose.
- Clone docker-main ecosystem from https://bitbucket.org/venkatnimmaturi/docker-main-ecosystem
- Clone myretail-product-manager from https://bitbucket.org/venkatnimmaturi/myretail-product-manager

- Clone myretail-elastic repository from https://bitbucket.org/venkatnimmaturi/myretail-elastic
- Import the cloned repositories to STS.
- Run "myretail-product-service" spring boot application in "Boot Dashboard". The REST API service should run on 8082 port.

# RESTful API Guide

## Overview

Here’s a quick guide to the RESTful endpoints offered by the myretail-product-service. At this time, you are able to Retrieve product info or Update price of the product in elasticsearch

## HTTP verbs

Profile tries to adhere as closely as possible to standard HTTP and REST conventions in its use of HTTP verbs.

| Verb | Usage |
|---|---|
| `GET` | Used to retrieve a resource |
| `PUT` | Used to update a resource |

## HTTP status codes

myretail-product-service tries to adhere as closely as possible to standard HTTP and REST conventions in its use of HTTP status codes.

| Status code | Usage |
|---|---|
| `200 OK` | The request completed successfully |
| `204 No Content` | An update to an existing resource has been applied successfully |
| `400 Bad Request` | The request was malformed. The response body will include an error providing further information |
| `404 Not Found` | The requested resource did not exist |

# Elasticsearch Guide

## Overview

Elasticsearch is a highly scalable open source analytics engine and documents store. It operates on REST protocol and query method is via JSON.

Index API is used to index or insert documents. Before indexing the documents it is a good practice to insert template mappings which pretty much defines the template of a particular index and the datatypes of fields in a document.

The following template is inserted before indexing products documents

PUT /_template/products-template
{
"template": "products*",
"mappings": {
"product": {
"properties": {
"id": {
"type": "long"
},
"price": {
"type": "nested",
"properties": {
"value": { "type": "scaled_float",
"scaling_factor": 100 },
"currency_code": { "type": "string" }
}
}
}
}}
}
=== GET and PUT API The search and update queries are also written in JSON. Although, the search object is composed using Jest Client library, under the hood it translates in to JSON.

Following are the search and update queries:

- Search GET /products/13860429/_search
- Update POST /products/product/13860429/_update
  {
  "script" : "ctx._source.price.value = 9.45;ctx._source.price.currency_code = \"EUR\""
  }

# Design

## Jest Client Common Library

```
This repository contains myretail-jest-common library. This library consists of common
code that can be used across the organization.
It configures Jest Client which can be used to communicate with elasticsearch.
Furthermore, code to perform health check on elasticsearch can be included in this
library.
```

## Product API

```
"myretail-product-api" artifact is aimed at holding the common API objects needed for
the communication with Restful webservices. The idea behind designing this as a
seperate artifact is reusability.
```

## Product REST Client

```
"myretail-product-rsclient" artifact is designed to communicate with another REST
service such as http://redsky.target.com/v2/pdp/tcin/13860428
This is designed such that any other service which needs to communicate for product
info can reuse this REST client. Moreover, the endpoint is made configurable. One more
reason is there might be other microservices which needs this library
and needed to be scale more depending on the needs.
```

## Product Jest Implementation

```
"myretail-product-jest-impl" is the actual implementation of jest code. Main
difference between Jest Client library (myretail-jest-common) and this artifact is
Jest Client library
configures objects needed for the communication of Elasticsearch where as this
artifact constructs actual queries to perform CRUD operations.
```

## Product Service

```
This artifact contains actual controller to accept REST requests. Since, the
requirement is to retrieve data from two sources, it performs retrieval in
multithreaded fashion for performance benefits.
```

## Design Patterns

```
Following design patterns are implemented to develop this application
```

- Dependency Injection
- Builder Pattern
- Template Design pattern

## Features

```
Following features are considered when designing this app
```

- Re-usability: The common libraries are developed such that boiler plate code can be reused.
- Scalability: This app can be scaled both horizontally and vertically based on the needs.
- Fault-tolerant: Hystrix code is included as a part of development. So, even the external dependencies are down, the MyRetail Product Manager microservice is fault tolerant and provides atleast deafult response.

# Recommendations to make this App suitable to go live

- Configure a configuration server to externalize all the application properties such as Rest Endpoint, Elasticsearch connection properties etc…
- Implement a discovery server so that other apps or microservices can communicate with this microservice using load balancing etc…
- Implement Hystrix circuit breaker design pattern ( code is already there, need to configure parameters such as threshold limit etc..)
- Since this microservice is aimed at providing data to outside clients, implement security.
- Create a docker image for this microservice so that it can be seamlessly installed across multiple environments.