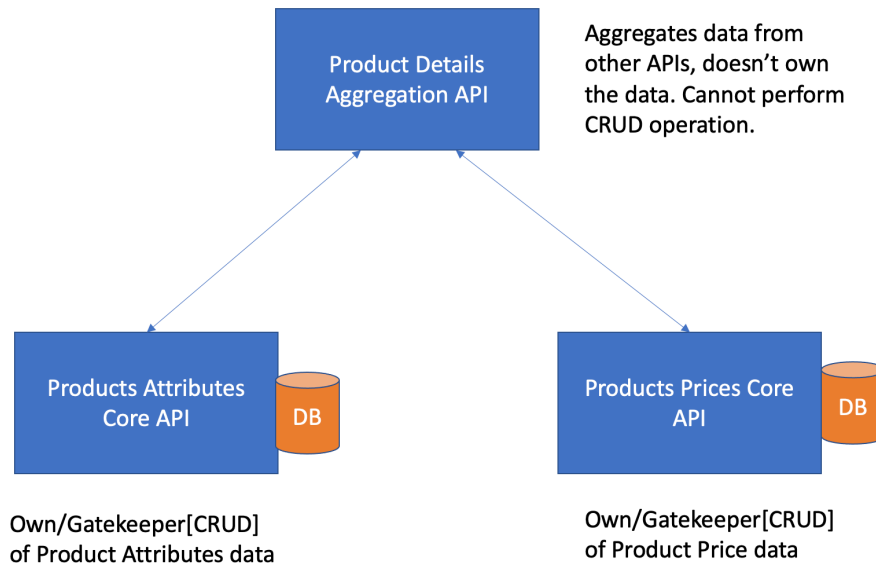


Architecture:



The requirement talks about one API which can 'render' price+details, and update prices.

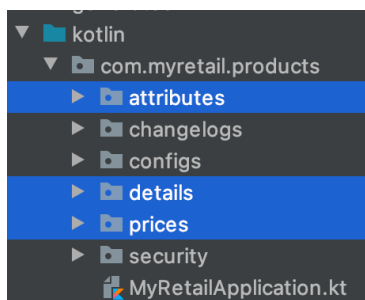
But I would like to follow the following principles, shown above.

Prices API – Core API responsible for everything related to a product's price. Is connected to an actual Datastore

Attributes API – Core API responsible for everything related to the name, department, unit of measure and so on and so forth. Is not connected to an actual datastore but renders dummy data. [This is the redsky simulated endpoint created for this case study]

Details API – Is an Aggregator API, which calls multiple Core APIs to render data, this could also end up being a Shared Service if needed. It will not Performs any Update or Delete operations. One must use the core APIs instead.

These three APIs functionalities are coded as 3 modules in the project, for review purposes. But in a real world scenario, they would be 3 different projects, hosted as their own API.



API Specification:

Please refer to My Retail API Spec: https://github.com/SeetharamanV/myRetail/blob/main/api-spec/myretail_spec-v1.yml

Paste the spec into <https://editor.swagger.io>, for you review

Callouts:

1. Key and Auth Token security enforced – Price is sensitive information, if anyone is trying to access it, they must pass in both.
2. Prices could have multiple variations, eg: Current price, Retail price or Initial price. Hence the put is to update a specific price_type, and returns the updated state of the entire Price Document related to the product id.

Technology and framework choices:

Please refer build.gradle file:

<https://github.com/SeetharamanV/myRetail/blob/main/build.gradle>

Building: Java 11, Gradle, Kotlin, Springboot, MongoDB + Mongock + Mongo Compass, Retrofit

Testing: Spock, Integration tests runs against a docker instance of Mongo defined here:

<https://github.com/SeetharamanV/myRetail/blob/main/gradle/docker-compose.yml>

Code coverage: Jacoco

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/build.gradle#L206>

Performance testing: JMeter <https://github.com/SeetharamanV/myRetail/tree/main/perf>

Manual Testing: Insomnia

Few questions you might have:

Why MongoDB?

Requirement stated to use a NoSql DB. And for this use case, I decided to go with Consistency + Partition Tolerance for prices, and hence decided on MongoDB.

Mongock is a way to managed: document schema versioning, indexes

Why Springboot?

It has nice easy addon to seamlessly integrate with MongoDB

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/kotlin/com/myretail/products/prices/repositories/PricesRepository.kt#L19>

MongoRepository is readily available for rapid development

Why Retrofit?

Robust way to integrate with partner API, it abstracts out the complexity of building the Request using the correct Url Properties, Adding the Key and Auth token to the headers. It keeps the service layer code cleaner and more maintainable.

Why JMeter?

Simple to use, it easily gets the job done.

Security:

Basic simple custom security setup

Created a simple set of Key and Bearer Token Request Filters, just for this case study, so that the application itself is self-contained for demo purposes.

Please refer:

Configuration:

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/resources/application.yml#L27>

Security folder:

<https://github.com/SeetharamanV/myRetail/tree/main/src/main/kotlin/com/myretail/products/security>

ApiKey Request Filter:

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/kotlin/com/myretail/products/security/ApiKeyRequestFilter.kt#L13>

Bearer Token Request Filter:

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/kotlin/com/myretail/products/security/BearerTokenRequestFilter.kt#L12>

Running the Code:

Basic setup and verification:

```
$ git clone https://github.com/SeetharamanV/myRetail.git
```

```
$ cd myRetail/
```

(Check Java 11 in classpath, and docker is up and running, before executing the next command)

```
$ ./gradlew clean build
```

(Make sure you get the following message. If yes, we can move ahead)

```
BUILD SUCCESSFUL in 1m 0s  
25 actionable tasks: 24 executed, 1 up-to-date
```

```
$ cd build/jacocoHtmlReport/
```

Open index.html for **code coverage information**

```
$ cd build/reports/tests/test/
```

Open index.html for **Unit tests Information**

```
$ cd build/reports/tests/integrationTest/
```

Open index.html for **Integration tests Information**

Executing the code:

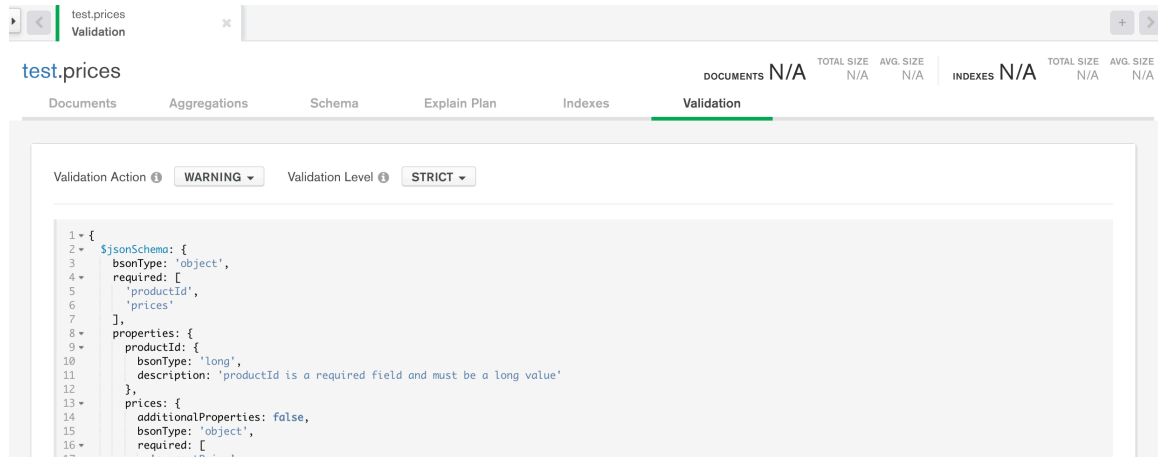
Prerequisite: MongoDB is running (either via docker or standalone)

Make sure you have the 'verified connection string' to the Mongo Database. It should be something like this

mongodb://username:password@localhost:27081/test?authSource=admin

```
$ cd myRetail/
```

```
$ ./gradlew bootRun --args='--spring.profiles.active=prod --  
spring.data.mongodb.uri=mongodb://username:password@localhost:27081/test?authSource=admin'
```

We see that after 'server startup', we have indexes on the productId, and schema for validating what is being saved in the collection.

This is done using Mongock, and configurations can be found here

Initial Schema:

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/kotlin/com/myretail/products/changelogs/DatabaseChangeLog.kt#L17>

Index on productId:

<https://github.com/SeetharamanV/myRetail/blob/59abfe7aff42602aa0f0d2fe9b788b00c6768d1f/src/main/kotlin/com/myretail/products/changelogs/DatabaseChangeLog.kt#L28>

Interacting with the application:

Make sure you have Insomnia or a rest tool of your choice. We are going to start.

=====

Attributes Core API:

Method: GET

Url: http://localhost:9090/my_retail/attributes/v1/products/1?excludes=department,unit_of_measure&key=TOKENONE

Header: authorization

Header value: Bearer `sometoken`

Response:

```
{
  "product_id": 1,
  "attributes": {
    "name": "Acme Glue"
  }
}
```

You can play around with excludes, the 3 recognized fields are name, department, unit_of_measure. If you pass an exclude other than that you will get the following 400

```
{
  "message": "Invalid attribute type passed for 'exclude'.",
  "code": "ATTRIBUTES-4000"
}
```

=====

Prices Core API:

Create some new prices for product Id 1:

Method: POST

Url: http://localhost:9090/my_retail/prices/v1/products/1

Header1: authorization -> Bearer `sometoken`

Header2: x-api-key -> TOKENONE

Post Body:

```
{
  "prices": {
    "current_price": {
      "value": "1.49",
      "currency_code": "USD"
    },
    "regular_price": {
      "value": "1.49",
      "currency_code": "USD"
    },
    "initial_price": {
      "value": "1.49",
      "currency_code": "USD"
    }
  }
}
```

Response:

```
{
  "product_id": 1,
  "prices": {
    "current_price": {
      "value": 1.49,
      "currency_code": "USD"
    },
    "regular_price": {
      "value": 1.49,
      "currency_code": "USD"
    },
    "initial_price": {
      "value": 1.49,

```



```
    "currency_code": "USD"
  }
}
```

If you try to POST again for the same product Id, you will get the following error:

```
{
  "message": "Prices document already exists for product id.",
  "code": "PRICES-4000"
}
```

GET prices for product Id 1:

Method: GET

Url: http://localhost:9090/my_retail/prices/v1/products/1?key=TOKENONE

Header1: authorization -> Bearer `sometoken`

Response:

```
{
  "product_id": 1,
  "prices": {
    "current_price": {
      "value": 1.49,
      "currency_code": "USD"
    },
    "regular_price": {
      "value": 1.49,
      "currency_code": "USD"
    },
    "initial_price": {
      "value": 1.49,
      "currency_code": "USD"
    }
  }
}
```

If there is no price data available this will be the response

```
{
  "message": "Couldn't find prices for the given product id.",
  "code": "PRICES-4040"
}
```

Update current price for product Id 1:

Method: PUT

Url: http://localhost:9091/my_retail/prices/v1/products/1/price_types/current_price

Header1: authorization -> Bearer `sometoken`

Header2: x-api-key -> TOKENONE

Put Body:

```
{
  "value":100.99,
  "currency_code":"USD"
}
```

Response:

```
{
  "product_id": 1,
  "prices": {
    "current_price": {
      "value": 100.99,
      "currency_code": "USD"
    },
    "regular_price": {
      "value": 1.49,
      "currency_code": "USD"
    },
    "initial_price": {
      "value": 1.49,
      "currency_code": "USD"
    }
  }
}
```

You can try the same for other regular_price and initial_price:

http://localhost:9091/my_retail/prices/v1/products/1/price_types/regular_price

http://localhost:9091/my_retail/prices/v1/products/1/price_types/initial_price

=====

Details Aggregation API:

Method: GET

Url: http://localhost:9090/my_retail/details/v1/products/1?key=TOKENONE

Header: authorization

Header value: Bearer `sometoken`

Response:

```
{
  "product_id": 1,
  "name": "Acme Glue",
  "current_price": {
```

```
"value": 100.99,  
"currency_code": "USD"  
}  
}
```

In the event price data is not available for a product id, the detail api will throw the following error

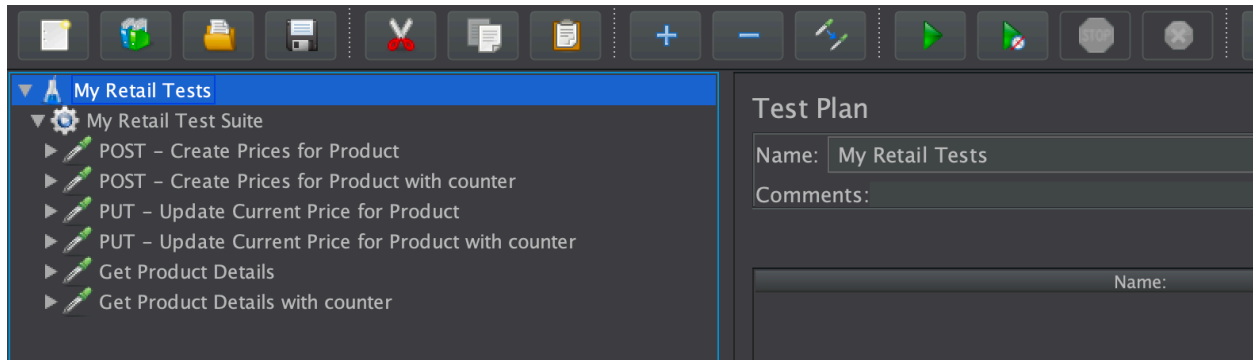
```
{  
  "message": "Couldn't find prices for the given product id.",  
  "code": "DETAILS-4040"  
}
```

A similar error will be thrown for unavailable 'attributes'

Performance Testing:

Make sure you have JMeter setup locally refer[<https://jmeter.apache.org/>]

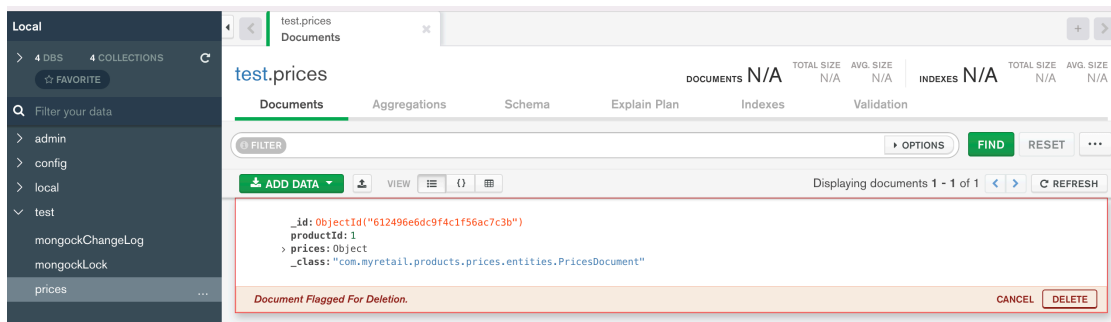
When JMeter is running locally, open the following file 'My Retail Tests.jmx' located in (\$ cd myRetail/perf) using the JMeter UI.



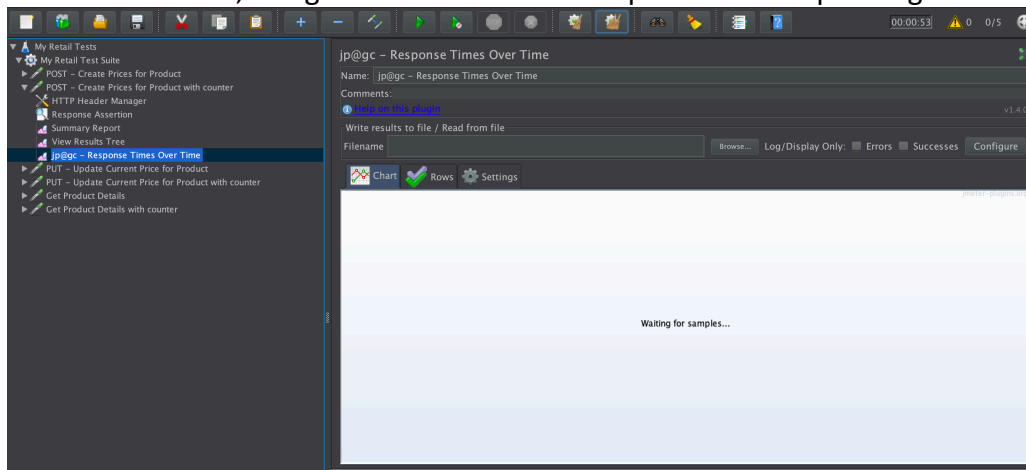
=====


POST - Create Prices for Product with Counter

Before starting to run the POST test, lets clear all the data out from Mongo, to avoid any 400 errors



Once that is done, lets get back to JMeter and open the corresponding test



And now we can start this test by hitting the green  button.

Results:



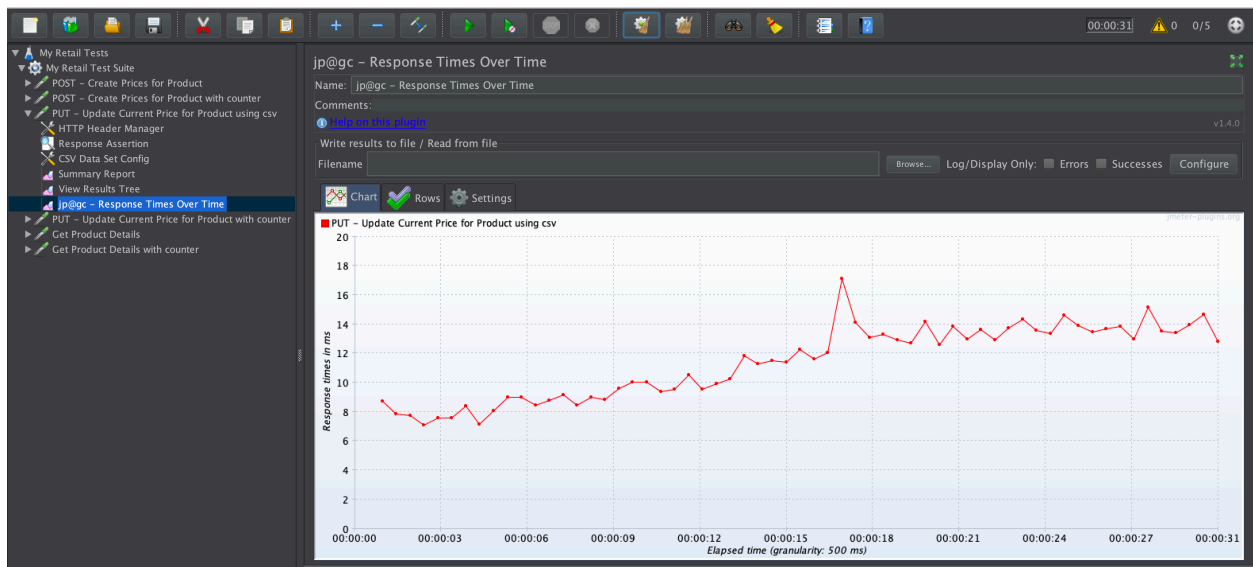
Response times: 4-7 ms

Highest: 22ms

=====

PUT - Update Current Price for Product using csv

Results:



Response times: 7-13 ms

Highest: 17ms

=====

GET Product Details using csv

Results:



Response times: 5.5-8 ms

Conclusion:

That is all for this document. Good day!