Import Catalog API

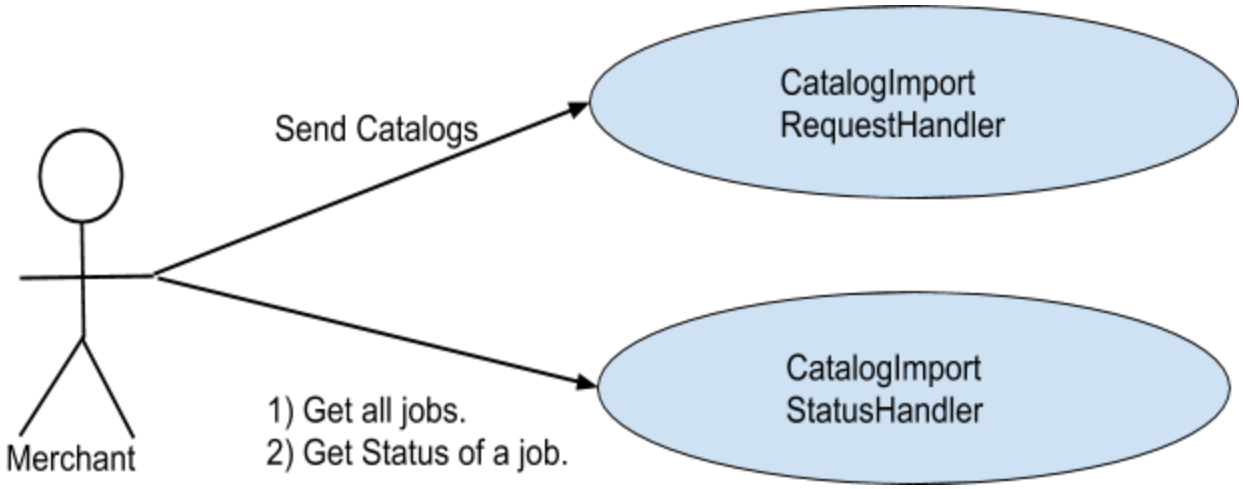
High-Level Technical Design

Version 0.1 Draft
04/14/2017
Author: Prabhu Chandrasekhar
Reviewers: Hanisha/Vikas Kumar

Overview:

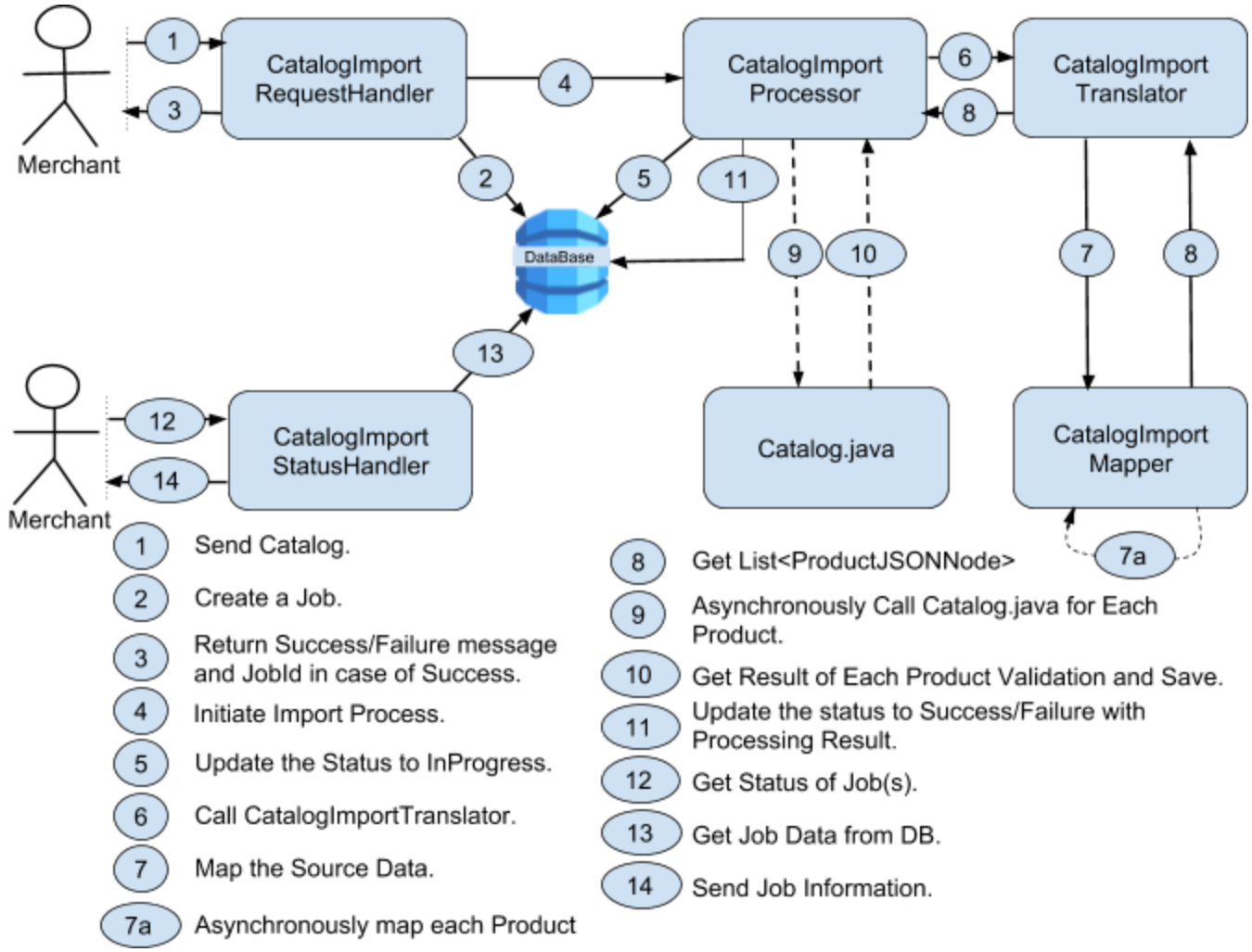
- This design document emphasizes the import of Catalog API into Store Inventory.
- The current implementation focuses on Google Catalog File(CSV/XML/JSON) and Store(CSV) file import. The design should be flexible/extensive to support future catering of other competitors (Shopify or BigCommerce) through YCC.
- The design should work as asynchronous communication meaning the Merchant should not be blocked after sending the Catalog file.
- The design and implementation should work independently on both MS1 and MS2. A flag could decide the flow to MS1 or MS2 for the below tasks.
    - Validation part
    - Database flow
    - Translator part

Use Case Diagram:

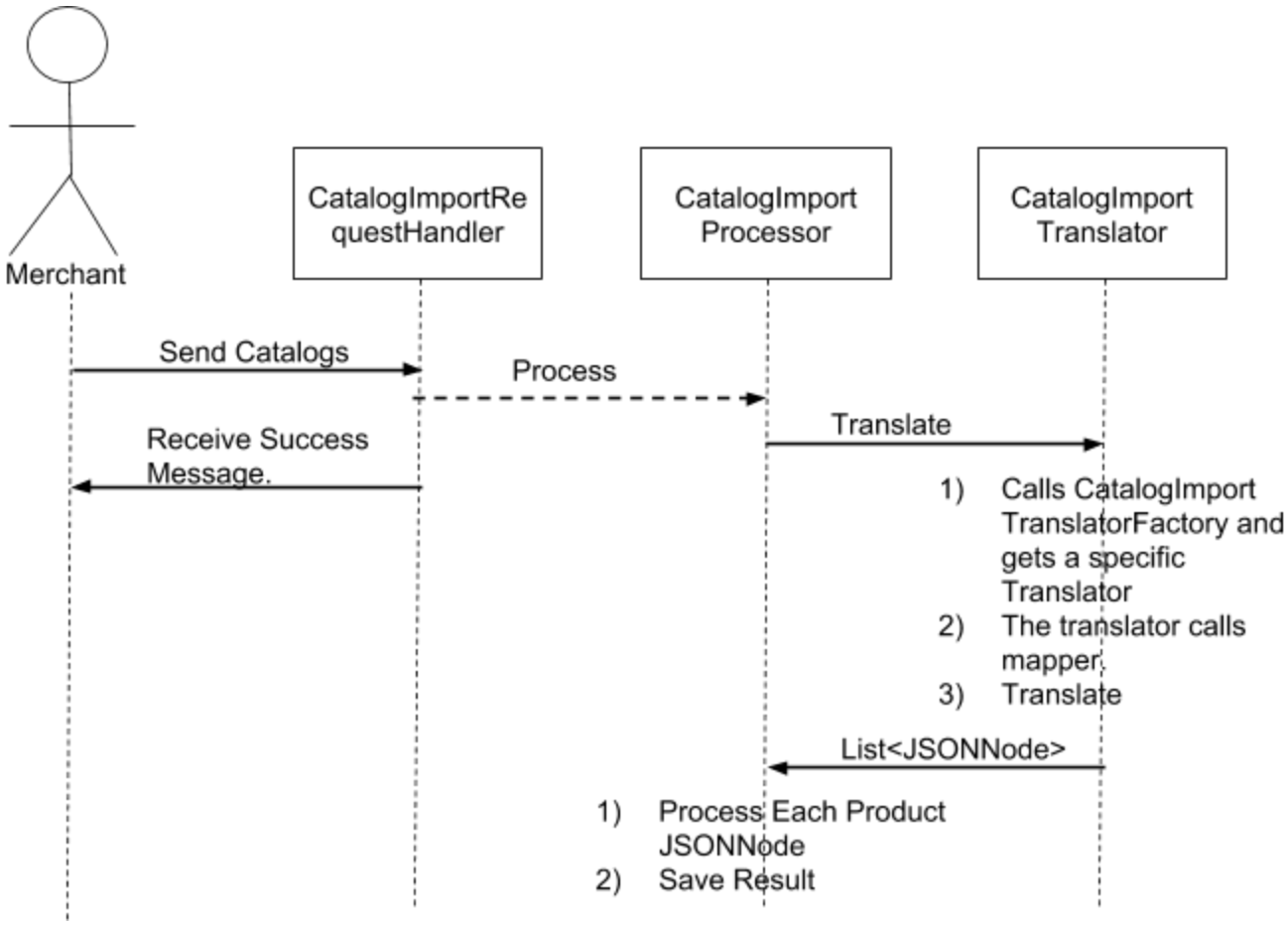


- Merchant can send catalogs through
    - file upload (CSV, XML, JSON).
    - Competitor site URL and Auth details. YCC will pick the catalogs, provide it to Catalog Import Request Handler through WS URL or file streaming.

Functional Diagram



Sequence Diagram



1) Merchant send Catalogs to Catalog Import Request Handler Service.
2) Catalog Import Request Handler receives the Catalog and does the following.
    a) Receives the Catalog, StoreId, format.
    b) Creates the JobId.
    c) If the catalog is received as a file, save file in AWS, gets back AWS URL.
    d) Creates an entry in CatalogJobs with StoreId, JobId, FileUrl, Status (Received) and format.
    e) Call Catalog API Processor.
    f) Send back success message with CatalogJobs pojo.
    g) Note: If the incoming file doesn't have a format associated. Leave the format column empty.
3) Catalog Import Processor does the following
    a) Receives the StoreId, JobId and FileURL.

b) Updates the CatalogJobs table Status field to InProgress.
c) Calls Catalog Import Validator to identify the file type. Format should be any one of JSON, XML or CSV.
d) Calls Catalog Import Translator and gets back List<JSONNode> of type Product.
e) Using CompletableFuture asynchronously call "Catalog.java" addProduct() method for each Product JSON.
    i) Validation and Processing (Save Product) on the data part will be continue as part of existing code base.
f) When all products are stored, updates CatalogJobs table with Number of Received products, Number of Products failed, failure reason URL and Status field to Completed.
g) Failed product reasons are gathered , stored in file and the file is uploaded to AWS.

4) Catalog Import Translator does the following
    a) Calls Catalog Import Translator Factory and gets a specific translator based on entity(Google/Shopify) and version. Then Translator calls mapper service.
    b) The mapper service loads the Catalog_Mapper table and does the basic field mapping and complex mapping and sends back List<Product>
        i) For image datatype, image_schema.json needs to be loaded. The resizable image needs to be created, upload it to AWS and the URL with size and other image data needs to be created and set in Product JSON. "option_attributes" and "parent_categories" are the other attributes which needs similar activities.
        ii) Fields like age_group, size, taxable, shippable etc… need transformation before setting to JSON.
        iii) Based on the variants (sizes[] + sizeType + sizeSystem + itemGroupId) multiple products entities are created.
        iv) Size attribute is specified, but default sizeSystem is missed. Ex: Clothing size 8, shoe size 8, handle it properly using Enum.
        v) Handle “unitPricingMeasure”, “unitPricingBaseMeasure” and “multipack”.
    c) Failed product due to data type mismatch and validation will be gathered and saved in AWS. File location is stored in “output_log”

Table Design

- Catalogs are stored in Dynamodb and table name is CatalogJobs.
- StoreId and JobId will be primary key for CatalogJobs table

- Status field can have any of the below values
    - Received
    - InProgress
    - PartiallyFailed
    - Success
    - Failed
- Catalog mapper information are stored in Catalog_Mapper table.
- “created_time”, “updated_time” and “locale” fields will be encrypted by default.
- store_id, id, status fields will be saved without encryption for easier understanding.

CatalogJobs

Column Name	**Data Type**	**Index**
store_id	String	Partition Key aka Primary Key
id	String	Sort Key aka Secondary Key. This is Job Id.

data	String (JSON)
	Example:
	{
	“input_uri”:”ftp://aws/file”
	“input_format”:”CSVorJSONorXML”
	“9jk”:”Google”,
	“version”:”1.0”
	}	

status	String	Index Key

meta	String (JSON)
	Example:
	{
	“failure_count”:”10”,
	“received_count”:”100”
	“output_log”:”https://awsFileUrl”
	}	

Catalog_Mapper

Column Name	**Data Type**	**Index**
api_name	String	composite partition-sort key

version	String	composite partition-sort key

data	String
	Example:
	[
	{
	"source_id":"id",
	"destination_id":"id",
	"source_type":"String"
	},
	{
	"source_id":"title",
	"destination_id":"display_name",
	"source_type":"String"
	},
	{
	"source_id":"imageLink",
	"destination_id":"images",
	"Source_type":"image"
	},
	{
	"source_id":"price",
	"destination_id":"list_price",
	"Source_type":"price"
	}
	]	

CatalogVariants

Column Name	**Data Type**	**Index**
base_id	String (Base Product Id)	Partition Key aka Primary Key
id	String (Option Product Id)	Sort Key aka Secondary Key. This is Job Id.
store_id	String	Index Key
deleted	Boolean	False by default
data	String(JSON) **Example:** { “variants”:”color, size”,	

	“products”:” Entire Product String ” }	

API List

1) Receive a Catalog

Request:

HttpMethod: POST

API URL: /catalogapi/v1/{storeId}/catalog?fileUrl = “AWS Url”

Payload Body {

data= “fileData”

}

Sample Request:

curl -X POST -d@/home/chandrasekhar/product.json "http://localhost:4080/catalogapi/v1/ys-50071731-2/catalog"

Sample Response:


```
{
  "msg": "Receive Success",
  "code": 0,
  "data": {
    "store_id": "ys-50071731-2",
    "job_id": "d7f4ddb9801e48a3bd486c74f6a1d0bc",
    "successCode": 0,
    "status": "Received"
  }
}
```


Note: Logic needs to be added to find the input format (CSV or JSON or XML). Read the file as Stream with Buffer to avoid keeping bulk data in memory.

.

2) Get Status of a job for a store

Request:

HttpMethod: GET

API URL:

/catalogapi/v1/{storeId}/catalog/job/{jobId}

Sample Request:

curl -X GET
"http://localhost:4080/catalogapi/v1/ys-50071731-2/catalog/job/d09f14c12dc6400189
72cf859fbd76ed"

Sample Response:


```
{
  "msg": "Success",
  "code": 0,
  "data": {
    "total": 1,
    "results": {
      "store_id": "ys-50071731-2",
      "update_time": "1513123764084",
      "data": {
        "input_uri":
"https://s3.amazonaws.com/store-catalog-import.qa.lumcs.com/
ys-50071731-2/_catalog_import_3043046d844649b591d9fb4b953892
af",
        "version": "1.0",
        "entity": "google",
        "input_format": "json"
      },
      "create_time": 1513123754578,
      "meta": {
        "output_log":
"https://s3.amazonaws.com/store-catalog-import.qa.lumcs.com/
ys-50071731-2/_catalog_import_3043046d844649b591d9fb4b953892
af_error",
        "received_count": 4,
        "failure_count": 2
      },
      "id": "d09f14c12dc640018972cf859fbd76ed",
      "locale": "en_US",
      "status": "Partially Failed"
    }
  }
}
```


3) Get Status of a last job for a store

Request:
HttpMethod: GET
API URL:
/catalogapi/v1/{storeId}/catalog/job?orderby=”create_time”&top=”1”
Sample Request:
curl -X GET
"http://localhost:4080/catalogapi/v1/ys-50071731-2/catalog/job/d09f14c12dc640018972cf859fbd76ed"

Sample Response:


```
{
  "msg": "Success",
  "code": 0,
  "data": {
    "total": 1,
    "results": {
      "store_id": "ys-50071731-2",
      "update_time": "1513123764084",
      "data": {
        "input_uri":
"https://s3.amazonaws.com/store-catalog-import.qa.lumcs.com/
ys-50071731-2/_catalog_import_3043046d844649b591d9fb4b953892
af",
        "version": "1.0",
        "entity": "google",
        "input_format": "json"
      },
      "create_time": 1513123754578,
      "meta": {
        "output_log":
"https://s3.amazonaws.com/store-catalog-import.qa.lumcs.com/
ys-50071731-2/_catalog_import_3043046d844649b591d9fb4b953892
af_error",
        "received_count": 4,
        "failure_count": 2
      },
      "id": "d09f14c12dc640018972cf859fbd76ed",
      "locale": "en_US",
      "status": "Partially Failed"
```

```
        }
      }
    }
```


4) Get all jobs for a store: Gets storeId as Input and returns list of job ids and their status.

Request:

HttpMethod: GET

API URL:

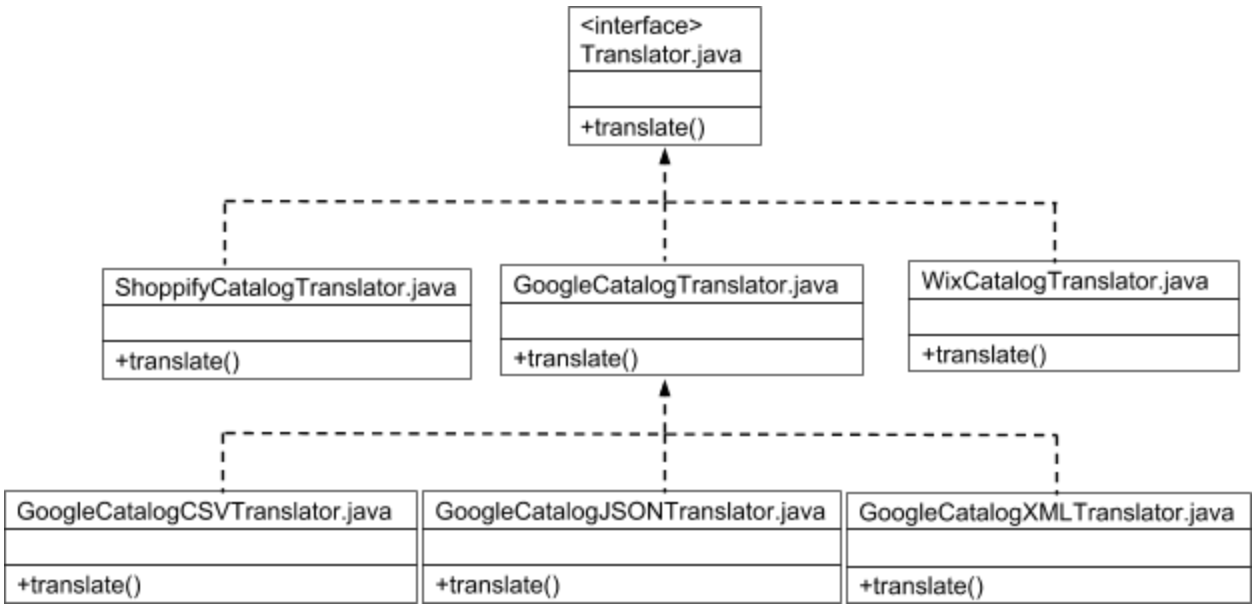
/catalogapi/v1/{storeId}/catalog/job?orderBy=updatedTime&startWith=1&endsWith=10

Response:


```
{
   "msg":"success",
   "code":200,
   "data":[
      {
         "storeId":"store1",
         "jobId":"job1",
         "data":{
            "input_uri":"https://aws/file"
         },
         "status":"Completed",
         "meta":{
            "failure_Count":"10",
            "received_Count":"100",
            "output_log":"https://aws/errorFile"
         }
      },
      {
         "storeId":"store1",
         "jobId":"job2",
         "data":{
            "input_uri":"https://aws/file"
         },
         "status":"Completed",
         "meta":{
            "failure_Count":"0",
            "received_Count":"100"
         }
      }
```

```
        ]
    }
```

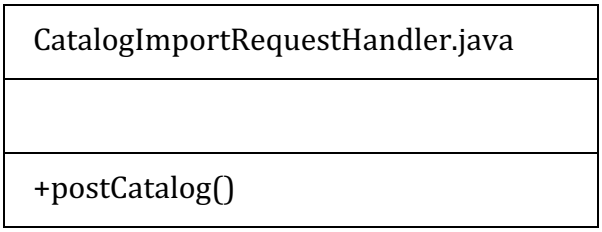
Support on Various Translators

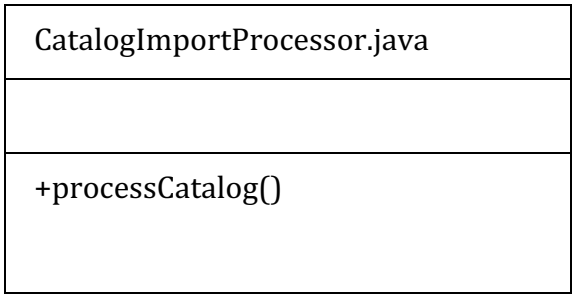


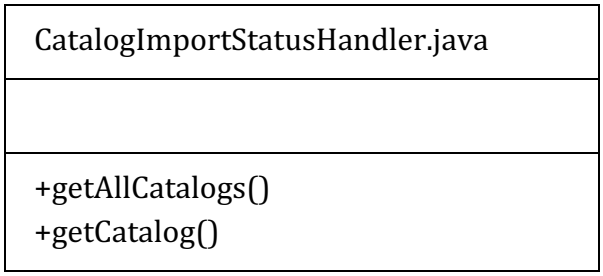
Implement Strategy design pattern to support various translator at run time.

Class Diagrams

- JAXRs Jersey Rest provider receives the JSON request.
- Package structure “com.yahoo.ysb.store.apis.catalog.import”
- Package structure “com.yahoo.ysb.store.apis.catalog.import.google”
- CatalogImportProcessor should follow Template Design Pattern to execute the flow.







Mapper Design

1) Load mapper data from Catalog_Mapper table.
2) Read Mapper Data Type as key value pair and place it in map.

   id=String
   title=String
   imageLink=image

3) Read Mapper as key value pair and place it in map. Key will be Google attribute and value will be Store attribute

   id=id
   title=display_name
   imageLink=images.id.url

4) Read incoming file/stream as key value pair.

   FasterXML- jackson-dataformat-csv: provides support for loading CSV file.
   FasterXML- jackson-databind: provides support for loading XML and JSON.

5) Compare the keys of mapper and incoming catalog file keys.

6) Replace incoming catalog key with store key based on comparison.
7) Read Store JSON Schema as ArrayNodes

```
ArrayNode productSchema = (ArrayNode) rootNode.get("fields");
Iterator<JsonNode> productSchemaIterator = productSchema.elements();
```

8) Create Product JSON structure with above schema and data in steps3. Before setting the data compare the data type in mapper with Schema, if it matches set the data. Otherwise additional processing may needed. Ex:
    a) For image datatype, image_schema.json needs to be loaded. The resizable image needs to be created, upload it to AWS and the URL with size and other image data needs to be created and set in Product JSON. "option_attributes" and "parent_categories" are the other attributes which needs similar activities.
    b) Fields like age_group, size, taxable, shippable etc… need transformation before setting to JSON.
9) Source Mapping document can be found here

Sample Data

CSV
JSON
XML

Assumptions/Constraints/Limitations

- The design should be extensible to provide pre-validation step for the incoming data. Google Catalog Implementation might skip this for now. This will help in 2 ways
    - We can capture the error sooner before processing.
    - Solve unknown error at processing, if the processing is happening without validation. Ex: Executing Curl command on the Image URL.
- If the Catalog file is imported again, then the product will be created or updated based on the “id” column associated with the each product. (Stash Pending)
- The data type on the mapper can be “Generic” or “Custom” rather than adding different data type. Implementation needs to take step on this based on pros and cons.
-
-

Questions

- Priorities on which one to load, when both AWS Url and File is uploaded. Now code throws error on this situation.

Implementation/Performance Considerations

- Import file can be big. So implementation should be made on how to load file at server which should not cause out of memory exception. Possible options are
    - Limit max file size in client side
    - Break the file on client side
    - Use multipart/form-data
- Streaming read to avoid performance issue of loading entire file as DOM

```
// NOTE: type given for 'readerFor' is type of individual row
MappingIterator<String[]> it = mapper.readerFor(String[].class)
   .readValues(csvSource);
while (it.hasNextValue()) {
  String[] row = it.nextValue();
```