## Purpose

To train any new software developer - fresher or lateral on Full Stack Development

## Project

### Overview

Create a **PoS (Point of Sale)** application. PoS systems are what you see in grocery stores, where the sales manager scans your products and give you a receipt.

### Functional Requirements

1. Upload brand/category details (Brand Master) using TSV, from UI
2. View + Create + Edit a brand detail using UI
3. Upload product details (Product Master) using TSV, from UI
   1. a product cannot be inserted if its brand-category does not exist in the Brand Master
4. View + Create + Edit a product detail using UI
5. Upload product wise inventory using TSV
6. Edit inventory for a product
7. Create a customer order
   1. Enter barcode, Quantity, MRP (multiple rows)
   2. On creating order, the inventory should get reduced
   3. An order should cannot be created for a product which does not exist
8. Edit an existing customer order
9. Download a PDF for a customer invoice (so that it can be printed)
10. Reports
    1. Inventory report
    2. Brand report
    3. Sales report (for a duration)

**Notes**

* All data should be stored in a small case, trimmed.
* User can enter mixed case values in all forms, but all comparisons internally must be done in lower case
* TSV stands for Tab Separated Values. It is similar to CSV (comma-separated), just that we are using tabs instead of commas
* For PDF creation, use **Apache FOP** library <https://xmlgraphics.apache.org/fop/>

### 

### Schema

#### Brand

* brand-category combination must be unique

|  |  |  |
| --- | --- | --- |
| **id** | **brand** | **category** |
| *int* | *string* | *string* |
| 1 | nestle | dairy |
| 2 | dabur | health |

#### Product

* barcode must be unique

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **barcode** | **brand\_category** | **name** | **mrp** |
| *int* | *string* | *int* | *string* | *double* |
| 3 | a1b2c3d4 | 1 | nestle kitkat 100 grams | 49.00 |
| 4 | 11sdfs23 | 2 | chyavanprash 500 grams | 599.25 |

#### Inventory

|  |  |
| --- | --- |
| **id** | **quantity** |
| *int* | *int* |
| 3 | 49 |
| 4 | 10 |

#### Order

|  |  |
| --- | --- |
| **id** | **time** |
| *int* | *datetime* |
| 1001 | 12-02-2019 12:45 |
| 1002 | 12-03-2019 11:56 |

#### OrderItem

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **orderId** | **productId** | **quantity** | **sellingPrice** |
| *int* | *int* | *int* | *int* | *double* |
| 100001 | 1001 | 3 | 4 | 49.00 |
| 100002 | 1001 | 4 | 1 | 599.25 |

### Reports

Report should have filters on date range, brand and category. It should output the sum of quantities and sale amount at the category level. Eg.

#### Sales Report

**Filters**

|  |  |  |  |
| --- | --- | --- | --- |
| **Start Date** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **End Date** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| **Brand** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | **Category** | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Output**

|  |  |  |
| --- | --- | --- |
| **Category** | **Quantity** | **Revenue** |
| chocolates | 212 | 20,201.20 |
| dairy | 102 | 10,115.00 |
| 100002 | 1001 | 4 |

#### Brand Report

Same as Brand table

#### Inventory Report

Same as inventory table, but with brand and category in it i.e.

#### 

|  |  |  |
| --- | --- | --- |
| **brand** | **category** | **quantity** |
| nestle | dairy | 10 |
| dabur | health | 11 |

### Resources

* You can find a boiler plate code at <https://github.com/increff/employee-spring-all>
* This boilerplate code will already have following implemented
  + Upload a file using TSV, from UI
  + Thymeleaf, JQuery, Bootstrap
  + 1-2 Reports
  + Spring Unit Tests, Swagger
  + **mvn clean install, mvn jetty: run**
  + Maven Surefire plugin code coverage

### Judgement Criteria

Your code will be judged on the following criteria.

1 marks for each of the following items.

Total marks 10

1. Having a proper Maven build
   1. build using **mvn clean install**
   2. run using **mvn jetty:run**
2. Proper usage of Spring, JQuery, Thymeleaf, MySQL
3. Submit your code using **GitHub** repository
4. All code except Controller must have automated tests
   1. 80% code coverage, 90% method coverage, 90% class coverage
   2. Tests for all layers - dao, api, utilities
5. Code must be properly layered. For e.g. there should be no business logic in Controller or Repository layer, there should not be any queries in Service layer
6. It should be a clean code
   1. No commented out code snippets
   2. Code must be properly packaged
   3. Be careful about when to use **public, private, protected, static** modifiers
   4. Be careful about using @Transactional properly. E.g. if 2-3 entities need to be stored in a transaction, they should be wrapped using @Transactional
7. UI should be clean and look professional
8. Limits - A malicious user can potentially upload a file with very large data. All file uploads must be limited to 5000 rows maximum
9. Error messaging - User should be shown proper error in the UI if there is some issue in the data being submitted via TSV or forms
10. There should be code comments explaining what the code is doing. This is specially required in your test code