jBilling Backend Practical Test

This new test is a more practical exercise, where you need to write some code. The requirements are fictional and simplified so you can complete this in a reasonable time, but keep in mind that **this is your chance to show us** how well you can implement requirements. Do your best!

The Prizy Pricer

The company ‘Prizy’ wants to start conducting surveys of prices for different products, and then based on the collected information, calculate an ‘Ideal Price’ for a product.

The idea is to have a number of workers walk into a store, and with their iPads access a web application where they enter prices as they see them. In real time, an administrator will be able to see the entered prices using the same web application and the application will provide an ‘Ideal Price’ for a specific product.

There will be 4 web service API for the backend.

The project scope is to design and code the Backend. Frontend/UI is out of scope of this project.

Product API

To manage products in the the system (i.e create, list, update, delete) This API will be used to create new products in the system. Following are the important attributes of the Product

- Product Identifier (unique system generated value)
- Product Name
- Base Price (price of the product as per manufacturer)

Store API

To manage stores in the system (i.e create, list, update, delete) . This API will be used to create new stores in the system.Following are the important attributes of the Store.

- Store identifier (unique system generated value)
- Store Name

Price collection API

This API is what the Frontend calls to save the pricing data for products which workers found at the different stores . Following are the important fields on this product :

- Store identifier (to id the store uniquely).This has to match a table with unique store identifier key. That table also has a name for the store
- Product identifier (to id the product uniquely). This has to match a table with unique product identifier key. That table also has a name for the product
- Store Price

Product Pricing details API

This API is used to retrieve the product pricing details given a identifier for the product. The product details provides following fields.

- Product identifier
- Product Name
- Base Price
- Average Store price
- Lowest Store Price. This is lowest price seen so far. This value should be always be the most updated.
- Highest Store Price. This is highest price seen so far. This value should be always be the most updated
- Ideal Store Price. The last calculated/cached value of ideal price for this product.
  Notes:
  "*Ideal Price calculation logic*" section contains details on **how** to calculate this price.
  "*Ideal Price calculation refresh*" section contains details on **when** to calculate/refresh this price
- Number of prices collected
- Anything else that might be useful?

Ideal Price calculation business logic

- This price is calculated by taking all the prices of this product, removing the 2 highest and 2 lowest, then doing an average with the rest and adding 20% to it. It is known that this complicated formula **will be changed often**.
- Even different installations of Prizy Pricer will be using different business logic of this ideal price formula. So the key here is to try to minimize the impact to the application when a new formula needs to be put in place (hint: use something like the 'Strategy' design pattern).

Ideal Price batch calculation logic

- **At regular frequency** we will have application logic code which calculates the ideal prices for all the products which have been entered so far. This is executed via some timer/scheduler mechanism.
- Given large numbers of products (possibly millions) and large number of entries for each product from different stores, ensure that batch logic runs fast and not load the database

heavily during its run. Hint (hint: Use efficient DB queries and concurrency mechanism of the programming language)
- Additional an API will be exposed to trigger this logic at any point in time. See Appendix section 5 for API signature.

API Signatures

Api signatures of important APIs are mentioned in the Appendix below.

What to implement

- Above mentioned API.
- Business logic to calculated the ideal prices , design it to be easily replaceable with another logic by just configuration mechanism
- Application logic to calculate the ideal prices at regular frequency

You do not need to bother with usernames and passwords (we don't need to know who did what, or any form of authentication).

Testing

You have to provide automated tests for this application. In fact, the first thing we'll evaluate is the quality of the testing in place. Make sure that the test pass! Most people fail here. If you need us to do something before running the tests (like initialize the DB), say so in some instructions.

Recommended Technology and Environment.

The Services (API) tier is recommended to be implemented in Java,you are encouraged to use frameworks like Spring, SpringBoot or just plain Servlets. If you are comfortable then you can use Groovy as well.

Business Logic and Application logic can be implemented using frameworks mentioned above where applicable. However It would be advisable to code the logic related to Pricing calculation and batching in Core Java using standard library classes and features.

The DB can be any of these: PostgreSQL(preferred)/HSQL/MySQL(acceptable). You can use any DB version, which is not too old. Use a suitable ORM library/framework to provide access to database and serving queries/updates.

Delivery

- A zip file with all the necessary files. Email this to abhishek.pandey@appdirect.com with the Subject “Java Practical Test”. Don’t forget to include your name to this email.
- A simple way to build the application (do not submit the binary).
- A simple way to run the tests.
- A DB dump with the database initialized, or some other way so we can use the application and see some data in it (products and prices for those products).
- Add to the zip file a README.txt file with any and all the instructions that we need in order to run the application and the tests.

Evaluation

- We do not expect you to have strong knowledge of various frameworks.
- We expect you to have the ability to write clean code using core language features
- We are more interested in how good you are at designing and implementing a piece of code from scratch. Remember to keep in mind SOLID and DRY design principles while implementation.
- Keep in mind the correctness of the code you write.
- Quality of the tests.
- Design to ease the implementation of new ideal price formulas.
- Runtime performance of batch calculation logic
- Use of right set of Data Structure, Design Patterns and Concurrency patterns and language features.

We would expect you to use frameworks where recommended and not use them where it is advised against , this is outlined in section “Recommended Technology and Environment”

Enjoy coding!

Appendix

Signatures of some important APIs.

1. POST - /store
   RequestBody :
   {
   name:<name>,

description:<description>

}

ResponseBody

{

status: 200,

message: “success”,

payload: {

name: <name>,

description: <description>,

created: <created-date>,

identifier: <unique-store-identifier>

}

}

2. POST - /product

RequestBody :

{

name:<name>,

description:<description>

basePrice:<base-price>

}

ResponseBody

{

status: 200,

message: “success”,

payload: {

name: <name>,

description: <description>,

baseprice: <base-price>

created: <created-date>,

identifier: <unique-product-identifier>

}

}

3. POST - /store/product

RequestBody :

{

store:<store-identifier>,

product: <product-identifier>,
price: <product-price>,
notes: <product-notes>
}

ResponseBody
{
status: 200,
message: “success”,
payload: {
store:<store-identifier>,
product: <product-identifier>,
price: <product-price>,
Created: <created-date>
}
}

4. GET - /product/{product-identifier}/prices
ResponseBody :
{
status: 200,
message: “success”,
payload: {
product: <product-identifier>,
name: <name>,
description: <description>,
basePrice: <base-price>,
averagePrice: <avg-price>,
lowestPrice: <lowest-price>,
highestPrice: <highest-price>,
idealPrice: <ideal-price>,
count: <no-of-entries-of different-prices-for-the-product>
}

}

5. POST - /jobs/pricecalculator?command=start
Response Header:
ResponseBody :
{
status: 202,
message: “success”,
payload: {
job: <job-name>
started: <started-date>

```
        }
    }
```