Problem Statement:

For this take-home coding challenge, please choose one of the prompts below. To provide further context, we would like to see code (e.g., classes and high level design). Your solution doesn't need to be a complete application, but the main overall classes to model the problem space and unit tests (there can be some pseudo code so that it isn't a full implementation). Please let me know if you have any questions!

Option A (back-end test):

"Design a Marketplace for internet plans. This marketplace will read in plan data from a json file. A plan might have a name, price, tier (residential, SMB, commercial), price per month, and total data usage. A User of the Marketplace should be able to sign up for notifications for plans below a certain price threshold.Implement the marketplace and any unit tests necessary to show your code is correct. Be prepared to talk about your overall design, any patterns used, and your reasoning behind the decisions that you made in your design."

Option B (front-end test):

"Design a Marketplace for cellular or internet data plans. This marketplace will read in plan data from a json source. A plan might have a name, price, tier (residential, SMB, commercial), price per month, and total data usage.A User of the Marketplace should be able to sign up for notifications for plans below a certain price threshold.Implement the marketplace and any unit tests necessary to show your code is correct. Be prepared to talk about your overall design, any patterns used, and your reasoning behind the decisions that you made in your design.Note: you may implement your marketplace in node.js or in a browser. It's your choice. A user interface is optional. We'd recommend using console.log() for any output to avoid spending too much time wiring up a web-based UI.Do not use any high-level frameworks such as Express, Angular, or React. You may use low-level libraries such as Underscore/lodash. If you choose to build a UI, you may use JQuery, but again, a UI is optional."

Solution:

Option A is designed and coded.

How to Run Server

(Requires JRE/JDK 1.8 and maven installed on the machine)

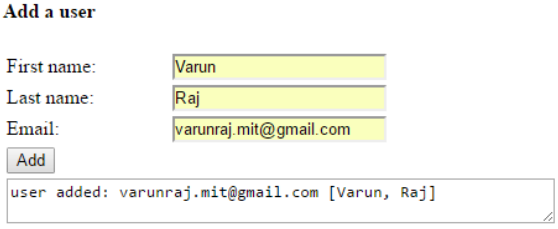
1. Unzip **test-app.zip** (included as part of submision)
2. In a command prompt go inside **test-app** folder where **pom.xml** file resides.
3. Run **mvn compile;**
4. Run **mvn test;** (Optional - required only run unit tests)
5. Run **mvn package;**
6. To start server: Run **mvn exec:java -Dexec.mainClass="Main"**

When server starts it reads **Internet Plans** from a file named **marketplacedata.txt** (provided as a part of submission – see **test-app/input** folder) containing json plan data. Application assumes to read this file from this location. Each time the server starts it logs the basic actions and API operations in a log file prefixed with current time stamp inside the folder **test-app/output**.

How to make requests

A very basic unintelligent html/javascript file has been provided to make REST requests to the server *running locally*. File assumes that server is API calls are accessible at http://localhost:4567/

1. Open **marketplace.html** (included as part of submission, folder **test-client**) in any browser (tested only with chrome).
2. To add/update **user** - Enter details, click **Add**

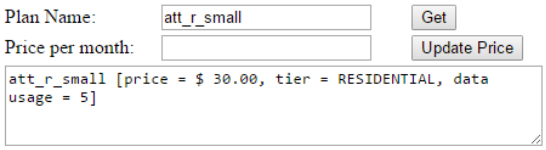


3. To signup a user - Enter details, click **Signup**

Signup user for notification

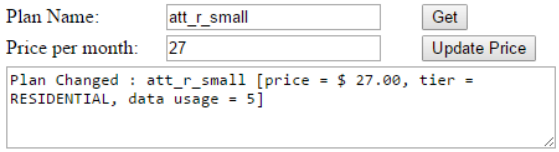
4. To get plan details (say plan price) – Enter plan name and click **Get**

Get Plan



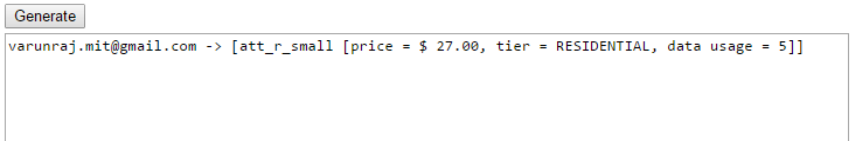
5. To update the price per month for a plan, enter plan name and price per month and click **Update Price**

Get/Update Plan



6. To generate all notifications in the system and clears out the notification system click **Generate**. Ideally this operation should have been performed by some backend system.

Genetare Notifications



7. Clicking **Generate** again should return no record.

<u>Assumptions around system requirements –</u>

1. **Market place** *only* has one conceptual plan known as **Internet Plan** – an **internet plan** can have **plan name, tier, price per month** and **data usage.** Question mentions something named **price** as well along with **price per month**, this concept has been dropped and only price exists in the system for an **Internet Plan** is **price per month**. A **plan** is identified by its **plan name** in the system. (Ideal world scenario should have used IDs in database instead of **plan name** probably)
2. **User** of this market place can have **email address**, **first name** and **last name**. A **user** is identified by its **email address** in the system. (Ideal world scenario should have used IDs in database instead of **email address** probably)
3. A **user** can *sign up* for a **plan** with a **Threshold** value.
4. A **user** should get notified whenever the **price per month** for a **plan** *drops* below the signed up **threshold** value.
5. A **user** can provide only one **threshold** value at one point for a given **plan**. He can surely be able to update his earlier **threshold** value for a **plan** at any point. Whenever **user** updates the **threshold** value for a **plan**, system clears out any earlier registered notifications from the system with old **threshold** value.
6. A **user** should be able to *sign up* for multiple **plans** in system at one point.
7. A **user** should get only one **notification** (a digest) for *all the **plans*** if more than one **plan** drops its **price per month** since the last time **notification** was generated.
   E.g. **User** u1 as signed up for **plan** p1 and p2 with **threshold** value t1 and t2 respectively. Supposed for both plans p1 and p2 **price per month** drops below t1 and t2 respectively. At this point **notification system** decides to

generate the notifications. It should generate *only one* **notification** to u1 with both p1 and p2 information into it. Now suppose p2 price again drops further and **notification system** decides to generate the notification at this point. This time it should again generate only one email but only for p2.
At some point suppose **notification system** runs again. This time it should not generate any notification as no price change happened to any of the signup plan *since the last time* it ran.

8. **User** should be able to "*Un Sign Up*" for any **plan**. Then user should not get any **notification** for the unsigned up **plan(s).**
9. It is the *change* in **price per month** (see API /planprice) for a **plan** which triggers the core logic of system, i.e. the registering of all the users (observers) for a plan (observable) with the **notification system**. – Observer pattern.
10. A separate system, **Notification System**, is supposed to generate **notifications**. Once ran it clears out all the generated **notifications** from the system.

System Limitations –

1. Adding a plan happens by reading plans from a json file when the system (server) starts. This is the only way plans can be added to system *as of now*. No RESTful API is provided to add a plan. Though the functionality has been provided to **Market place** and tested by integration test using Junit.
2. No RESTful API is provided for "Un Sign Up". Though the functionality has been provided to **Market place** and tested by integration test using Junit.
3. No database or persistence of any data is provided, so if a server stops or crashes all the previous data/state/actions will be lost. This is done purposefully not to have any data base dependency for this coding challenge.
4. Ideally system should be designed with support of database and persistence, but I believe scope and effort of problem would have increased drastically. I agree that in presence of database, design could have been completely different.
5. Only one API interface is provided for all actions like adding a user and generating notifications. Ideally in real world this could have been handled separately, especially it doesn't look good to have API to generate notifications.
6. System only tested and ran on Windows machine. Once server is started APIs can be accessible at http://localhost:4567/
7. Testing is not performed for scalability.
8. Unit test coverage is not provided for all the classes due to time constraint. Unit test framework is used for testing system logic but mostly are integration tests. Because of time constraint and unfamiliarity with JMokit I was unable to complete the 'perfect' unit testing using mocking. Irrespective, code coverage should be 100% for this submission.

System Design:

Verbs:

System has been visualized in terms of RESTful APIs. See attached SampleRequet.txt. System can:

1. *Add* an **InternetPlan** (RESTApi support is not provided for this, but tested via unit test framework)
2. *Add/Update* an **User** (see SampleRequest.txt)
3. *SignUp* an existing **User** for an existing **InternetPlan** for a **Threshold** (see SampleRequest.txt)
4. Un SignUp an already signed up user for a plan (RESTApi support is not provided for this, but tested via unit test framework).
5. *Get* plan's data and *Update* a plan's price per month. This is the triggering action which registers relevant users to notification system (see SampleRequest.txt)
6. *Get information* of all the notifications in the system (RESTApi support is not provided for this, but tested via unit test framework)

7. *Get information* of all the notifications in the system for a User (RESTApi support is not provided for this, but tested via unit test framework)
8. *Generate* and *Clears* all notifications from the system (see SampleRequest.txt)

Components / Classes:

Components are not designed in terms of Interfaces or abstract classes. Based on my assumptions of problem scope I decided not to introduce these. Every class is concrete in nature. Singleton pattern basically because there is no database support.

1. **Main**: Definition of service end points. **sparkjava web framework** (http://sparkjava.com/) to have a light weight REST based system in place. You need to have Java 8 installed (needs java path variables configurations).
2. **MarketPlace**: (Singleton) Maintains market place data – basically Users and Plans. This is also the main entry point of all logical operations (APIs) in the Market place system.
3. **InternetPlan**: Encapsulate an internet plan. Encapsulates all the user who are signing up for a concrete plan.
4. **User**: Encapsulates **user** information.
5. **NotificationSystem**: (Singleton) maintains all the registered notification in the system at one point. This information is maintained at per user basis.
6. **Tier**: Just an enum to represent **Tier** of an **internet plan**. Currently it is playing no role in any logic in the system.
7. **Money**: Encapsulates the **money** in the system. Helps in representing **price per month** or **threshold**. ***Code for this is taken from internet.***
8. **AppConfig**: (Singleton) – Reads application configuration from the **resources/config.properties** file.
9. **Logger**: (Singleton) – For logging purpose
10. **AppUtil**: Couple of util functions.