**Payments Application Design:**

The application is designed based on following assumptions and details:

1. When a price update is made, the application records information about the price update, but does not apply the new price to customer immediately.
2. The customer is notified 30 days before their next billing date and the new price takes effect from their next billing cycle. For this reason, the payment app stores the nextBilldate for every customer or needs to have this available via an API call.
   1. the number 30 would be configurable
   2. there may be different ways to notify the customer and we may want to receive some kind of acknowledgement before applying the price increase
   3. A scheduler runs this component every day at a specific time, to send out notifications exactly 30 days before the customer’s next billing date.
3. Notifications sent out may fail – we need to retry or try alternate mechanisms (this is not covered in my implementation)
4. We may want to try applying the update to only a select subset of customers within a given country and plan (possibly to get some metrics on existing subscriptions and impact on future subscriptions). Based on this, we may want to go ahead with the increase or not. The mechanism for doing this is not covered in this design.
5. When multiple price updates are made, the latest update will apply.
6. Assumption – a price update is effective immediately (we don’t need to wait to start processing it). Ideally we may have an effective date to indicate that we want to start processing this price update sometime in the future)
7. It is possible to make every processing step (as shown in the design.pdf diagram) a separate component and have them call each other via API calls.
   1. Advantage – loosely coupled. We can scale by adding more instances/vms running these components as needed. When number of customers increase, we can have the following options of scaling
      1. A known number of machines running the different components – each dedicated to serve a specific subset of customers (configurable).
      2. Dynamically calculate how much processing is needed and start components based on load. Use a load balancer to manage distribution of customers among different workers.
   2. Disadvantage – each component could fail for different reasons and we need to cover for them.

The following Objects have been used:

1. Plan – represents the streaming plan Netflix offers
   1. planCode (key)
   2. planName
2. Country – represents the countries supported (not implemented. instead just used country code to represent the country)
3. PriceInfo – pricing details for a given plan in a given country (example – pricing details for plan 1S in USA is represented by this object)
   1. planCode (key)
   2. countryCode (key)
   3. price
   4. status – (ACTIVE, OBSOLETE)
   5. rolloutStatus – (PENDING if priceUpdate is being processed. COMPLETE if all customers on this plan/country have been processed)
4. Customer – Represents the customer info.
   1. countryCode
   2. customerName
   3. email (key)
   4. nextBilldate
   5. onboardDate (not used for impl)
   6. planCode
   7. priceId
   8. status – ACTIVE (only currently ACTIVE customers will be impacted)

The above objects will have REST APIs to add/modify or delete information. However, only those needed for the flow have been implemented.

Other classes:

CustomerRepository – Persistence for Customer

PriceInfoRepository – Persistence for PriceInfo

CustomerListChunkMsg – contains a list of customer ids and their new price. This message is inserted into “priceupdate” topic. Many listeners may be listening. They pick this chunk and process price update for their list of customers.

PriceChangeNotificationMsg – contains new price and customer contact info. This message is inserted into “priceupdatenotify” topic by price update processor component. Notification listeners may pick it up and send out notifications based on business logic/customer preferences

PriceChangeAckMsg – contains acknowledgement from customer. Notification processors put this message. Ack-Processing component will listen to this message and update the customer table with the new effective price (which will apply from nextBilldate)

PriceInfoController

1. REST end point for making price update
2. Also contains another end point to call PriceUpdateChecker

PriceUpdateChecker

1. Checks if a price update is available (runs everyday) and processes updates available.
2. Retrieves IDs of customers impacted, breaks them into chunks (size of chunks is configurable based on latency of PriceUpdateChunkProcessor)

PriceUpdateChunkProcessor

1. Takes each chunk and processes all customers within the chunk.
2. Puts messages into queue for notifying customers

PriceChangeNotificationReceiver

1. Notify customer
2. Send acknowledgment message to queue

PriceChangeAckReceiver

1. Updates the customer to new pricePlan

**Note:**

For this implementation

1. H2 DB – database (data.sql contains sample data inserts)
2. activeMQ – messaging
3. Sample data contains:
   1. 1S – 10 customers
   2. 2S – 5 customers
   3. 4S – 4 customers

nextBilldate for these customers is either 30 days from today, 28 days from today, or 32 days from today.PriceUpdateChecker (which would ideally be run by scheduler), can be invoked using a POST as described in the section below.

**Access the app:**

<http://localhost:8080/my-app1-0.0.1-SNAPSHOT>

(displays Hello World!)

**Embedded DB (H2) console:**

<http://localhost:8080/my-app1-0.0.1-SNAPSHOT/h2-console>

(make sure JDBC URL is set to: **jdbc:h2:mem:testdb**)

username: **sa**, no password

click connect and browse through sample data

**Update price of one plan for one country**

POST: http://localhost:8080/my-app1-0.0.1-SNAPSHOT/priceinfo

Request Body:

{

"planCode":"2S",

"countryCode":"1",

"price": 13.99

}

**Start Price Update Checker (may be scheduled or called as needed)**

POST: <http://localhost:8080/my-app1-0.0.1-SNAPSHOT/schedulePriceChecker>

Request Body:

2018-06-06

**Observe log messages and updates in the DB**

Log file - logs/mylogfile.log