Problem Statement:

"Netflix rolls out price changes for our service periodically. In order to accurately and effective change the prices of 100M+ customers, we need to have a systematic solution for changing prices."

Overview

This document provides the overall approach taken in designing the solution for above mentioned functionality/use case. It will mention about system/database entities used for representing the information in database as well as passing information to/from end users/client systems to our system/service. It will also list out the REST APIs provided to end users (eg: Netflix system admins) or other internal applications for various operations.

Design Approach

As a global online streaming provider, it is common for company like Netflix to change its price from time to time. We would want to maintain a history of all the previous prices so we have a complete record of all the price changes performed so far. Furthermore since these price changes have direct impact on business (revenue, profit margins, new customer growth) and customer behavior (increased content streaming, changing/terminating service plan etc), we would want to keep the history of all old/past prices so we can link this information to other information for performing such analytics.

In this solution, I have considered following entities/tables for the system.

Country: Contains country information where Netflix does its business
ServicePlan: contains information about various service plan offered by Netflix
CustomerAccount: contains information about customer himself (like name, address, email address etc)
CustomerSubscription: contains information about service plan subscribed by customers (past and current)
PriceByPlanCountry: contains information about price for each plan and country (past and current)

The main table we work with in this solution is **PriceByPlanCountry. Country** and **ServicePlan** are just as an example of metadata that would be maintained for Country and 3 Service Plans. **PriceByPlanCountry** refers to these 2 tables by countryId and servicePlanId respectively.

CustomerAccount is an example of some table that would keep customer's profile information. This is not needed by **PriceByPlanCountry** table.

CustomerSubscription contains information about which customer has which service plan at present as well as in past (with start and end date for each plan). **PriceByPlanCountry table** doesn't need **CustomerSubscription** table. However if we decide to store current effective price for each customer (based on service plan and customer's country), **CustomerSubscription** will need to be updated whenever **PriceByPlanCountry** is updated. We consider updating **CustomerSubscription** table outside the scope of this solution. Please refer to section 'Other Considerations/Enhancements' for rationale behind this decision.

Database schema

PriceByPlanCountry table contains following fields:

Field Name	**Field Type**	**Description**
priceId	long	Unique identifier for this record, automatically generated by the system on insert
countryId	int	ID of country from Country table
servicePlanId	int	ID of service plan from Service Plan table
effectiveFrom	Date	Date when this price is effective from
lastUpdated	Date	Date when this record was last updated by system
price	double	Price that was rolled out for this countryId, servicePlanId on effectiveFrom date
isActive	boolean	true for currently effective price. Set to false by system when a new effective price is rolled out

Assumptions used

I have made following assumptions to ensure that system has only 1 active price for any (country, service plan) pair at any time as well as prevent any intentional or accidental manipulation of previous price rollout. Therefore system performs following checks during insert/update and delete operations via REST API.

1) Can't insert/update or delete price information for any of the past price roll outs (ie. with effectiveFrom date < today) to avoid manipulation of previous price rollout
2) There will only be 1 record for each unique country, service plan and effectiveFrom combination.
3) Active should be true for any price insert/update executed by APIs. active is set to false by system when a new effective price is inserted on go-live day (effectiveFrom==today).
4) Can't insert price information for future rollout (ie. with effectiveFrom date > today) since this may lead to undesired scenarios like we may have more than 1 active price for same (country and service plan) pair after a date >= effectiveFrom date
5) priceId information is ignored for insert (PUT API) operations as this ID is automatically generated by the backend system to guarantee unique priceId for each unique country, service plan and effectiveFrom combination.
6) For both Bulk price update, and single price update (by priceID), system ignores all fields except priceId, price and active field.
7) Insert/Update/Delete is only allowed for the price that is rolled out with today's date (i.e, effectiveFrom==today).
8) Price field should contain a value > 0
9) servicePlanId field should contain a value 1-3 (corresponding to 3 service plans)

Operations allowed

Our system allows following operations via REST APIs (please see section 'REST APIs' for further details)

1. Get information about current and past price rollouts
   We provide 3 different APIs to retrieve information for active, inactive or both at different levels (a) all countries and service plans, b) all service plans for given country, c) for a particular country and service plan)

2. Insert new price on go-live day ie., price with effectiveFrom same as the day of insert operation (POST API). When a new price is inserted, system automatically sets active field to false for previous effective price.

3. Update today's price

4. Delete today's price. When today's price is deleted, system automatically reactivates previous effective price (ie., set its active field back to true).

Example of new price rolle out

Let's say that our last price change for countryId=1 and servicePlanId=3 was made effective on Feb, 12 2017 and our system has a record for same as following

```
{
  "priceId": 19996,
  "servicePlanId": 3,
  "countryId": 1,
  "effectiveFrom": 1486800000000, → corresponds to 2017-02-12
  "lastUpdated": null,
  "price": 14.99,
  "active": true
}
```

If we want to roll out a new price on Jan 1st, 2019, we should insert a new record using our POST API as below (*Note: operation should be performed on Jan 1st, 2019)*
POST http://localhost:8080/v1/price

```
[
  {
        "active": true,
        "countryId": 1,
        "effectiveFrom": "2019-01-01",
        "price": 17.99,
        "priceId": 0,
        "servicePlanId": 3
  }
]
```

This will create a new record into the system and return the url http://localhost:8080/v1/price/19997 in location field of Response header as below

Response headers
content-length: 0 date: Thu, 27 Dec 2018 22:45:06 GMT
location: http://localhost:8080/v1/price/19997

REST APIs

Service provides following REST API endpoints to end users or client applications.

GET http://localhost:8080/v1/price

Example

- Get all prices for all countries and service plans
  http://localhost:8080/v1/price
- Get all previous (inactive) prices for all countries and service plan
  http://localhost:8080/v1/price?active=false
- Get currently active/effective prices for all countries and service plan
  http://localhost:8080/v1/price?active=true

GET http://localhost:8080/v1/price/countries/{countryId}

Example

- Get all prices for all service plans for countryId=1
  http://localhost:8080/v1/price/countries/1
- Get all previous (inactive) prices for all service plans for countryId=1
  http://localhost:8080/v1/price/countries/1?active=false
- Get currently active/effective prices for all service plans for countryId=1
  http://localhost:8080/v1/price/countries/1?active=true

GET http://localhost:8080/v1/price/countries/{countryId}/plans/{servicePlanId}

Example

- Get all prices for countryId=1 and servicePlanId=3
  http://localhost:8080/v1/price/countries/1/plans/3
- Get all previous (inactive) prices for countryId=1 and servicePlanId=3

http://localhost:8080/v1/price/countries/1/plans/3?active=false

- Get currently active/effective price information for countryId=1 and servicePlanId=3
  http://localhost:8080/v1/price/countries/1/plans/3?active=true

POST

- http://localhost:8080/v1/price
  This API allows bulk insertion (insertion of 1 or more new price). Only allowed for today's price (i.e., effectiveFrom same as insertion date).

  For bulk insert, if 1 or more price record fails to meet required validation criteria, system will send HTTP Status code 207 and a payload with specific error message, status code and original input for each of invalid input. Location field of Response header will include link for all successfully inserted prices.

PUT

- http://localhost:8080/v1/price
  This API allows bulk update (update of 1 or more existing price). Only allowed for today's price.

  For bulk update, if 1 or more price record fails to meet required validation criteria (example: priceId doesn't exist or priceId is for past date or price is < 0) , system will send HTTP Status code 207 and a payload with specific error message, status code and original input for each of invalid input. Location field of Response header will include link for all successfully updated prices.

- http://localhost:8080/v1/price/{priceId}
  This API allows update of single price record. Only allowed for today's price.

DELETE

- http://localhost:8080/v1/price
  This API allows bulk delete (delete of 1 or more existing price). Only allowed for today's price.

  For bulk delete, if 1 or more price record doesn't exist or is for past price, system will send HTTP Status code 207 and a payload with specific error message, status code and original input for each of invalid input. Location field of Response header will include link for all successfully deleted price.

An example of bulk delete is provided below. The output reports error about 3 prices (19991, 19996, 19997) in response body and success for 19998 in location header.

Input:

```
[
 {
        "priceId": 19996
 },
{
        "priceId": 19991
 },
{
        "priceId": 19997
 },
{
        "priceId": 19998
 }
]
```

Output:

Response body

```
[
 {
  "message": "Insert/Update/Delete operation not allowed for past (with effectiveFrom < today)",
  "status": "BAD_REQUEST",
  "inputEntity": 19996
 },
 {
  "message": "No price information was found with Id = 19991",
  "status": "NOT_FOUND",
  "inputEntity": 19991
 },
 {
  "message": "No price information was found with Id = 19997",
  "status": "NOT_FOUND",
  "inputEntity": 19997
 }
]
```

Response headers:
content-type: application/json;charset=UTF-8 date: Thu, 27 Dec 2018 23:19:07 GMT

location: http://localhost:8080/v1/price/19998 transfer-encoding: chunked

- http://localhost:8080/v1/price/{priceId}
  This API allows delete of single price record. Only allowed for today's price.

Sample data

Application preloads all 5 tables using some sample data. It loads 4 sample prices in price table using src/main/resources/sampledata/price_by_country_and_plan.csv. Data in this file is assumed to be valid (ie., only 1 active price for a given country and service plan) and no validation is done during the loading of this file. Use http://localhost:8080/v1/price to see all these preloaded price after launch of the application.

Technology/tools used for development

1) Spring boot application - Spring boot framework provides embedded application server(tomcat). It also manages dependencies automatically and cuts down the amount of boilerplate code needed to write database access for read/write, mapping database tables to POJOs, implement REST API and generate swagger documentation/UI.
2) In memory H2 database

Other Considerations/Enhancements

CustomerSubscription modification:

Earlier I had mentioned that modification of **CustomerSubscription** was kept out of the scope of this solution. However, let us say that we also decide to keep price information in **CustomerSubscription** table in addition to master/reference table **PriceByPlanCountry**. Below are pros and cons of doing this.

Pros: Keeping price information in **CustomerSubscription** is useful as we don't have to refer to master table every single time we need to find price, customer is being charged for his current subscribed plan (or was charged for past subscribed plans).

Cons: Keeping price information in **CustomerSubscription** will require us to update this information for all the affected customers each time new price is rolled out and updated in master **PriceByPlanCountry** table and need to be in sync at any point of time.

For the price rollout solution I have not considered updating **CustomerSubscription** for following reasons.

a) We want to follow microservices architecture where one service provides only 1 unique capability. The price service should only be providing the APIs to read/write/update and delete reference price (applicable for a specific plan in specific country) information.
b) There should be a separate microservice for interacting with and updating **CustomerSubscription** table. This microservice can use GET REST APIs provided by price service to get latest price information and update price information in **CustomerSubscription** appropriately.
c) Keeping separate micro-service for separate functionality/capability allows us to scale and upgrade each service independent of each other.

Other things not covered in my implementation but should be considered in a real solution.

Security: We should implement authorization scheme such that access to these APIs are limited to only authenticated users with appropriate authorization specially POST/PUT/DELETE APIs should be restricted to users with admin access.