

Context

Netflix offers customers 3 service plans (1S, 2S, & 4S) based on the number of concurrent streams and priced accordingly. As we are a global service, we have defined prices for each service plan for country we support. For each customer we store the service plan (1S, 2S, or 4S), the price for the chosen plan and the country of the customer.

Problem Statement

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

Objective

Design and implement a system that hosts Netflix pricing which will enable us to systematically change prices across all our global customers. We want to see how your system supports price changes pushed by country.

Core Use-cases

- Provide APIs to View & Modify Price of a Service Plans globally
- On Effective date of the price change, systematically update price across global customers
- effective, update all customer records who are affected by the price change with the right value considering the time zone. (Note: Alternatively, we could store the price & looked up during the billing cycle, it is clear that this is not an option to the problem we have in hand) - As there are more than 100+ Million Customers, take this scaling aspect into account

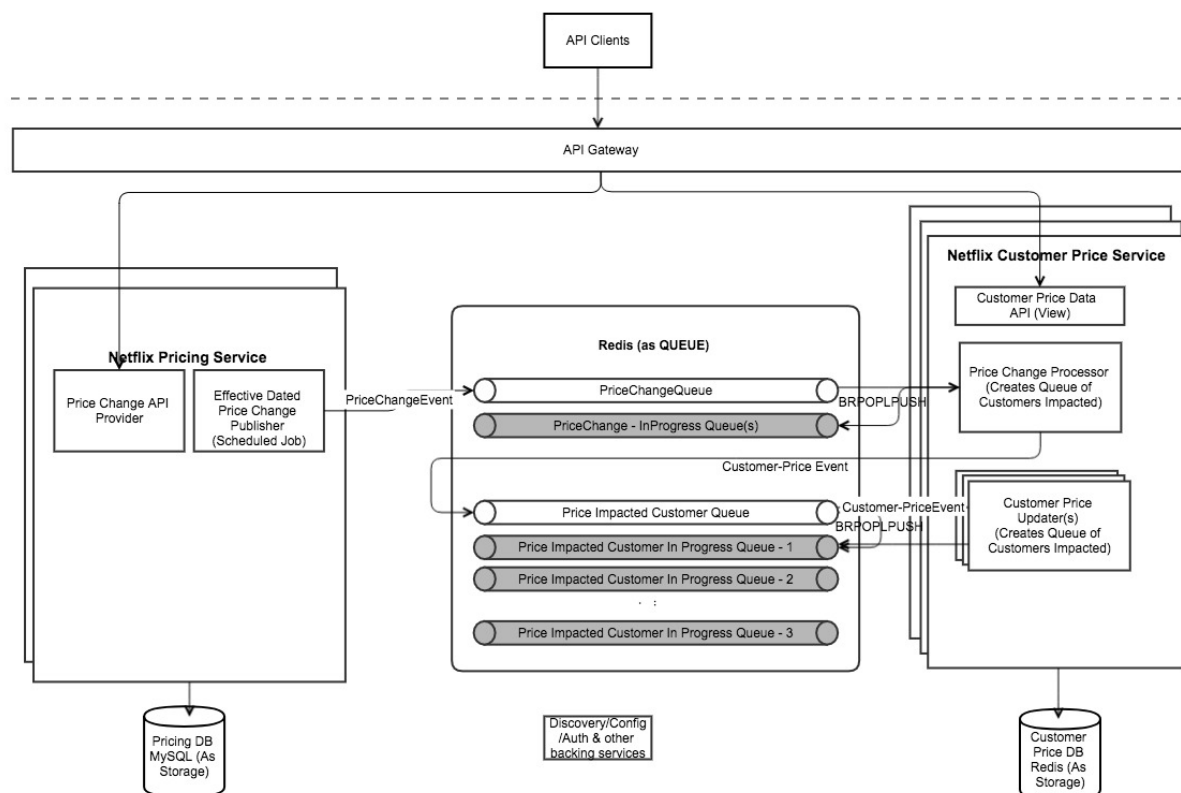
Considerations

- Volume: 100+ Million customers
- Effective Dated
- Ensure all customer prices are updated in timely manner to prevent any revenue loss
- Fault tolerant and resilience built-in as it's a critical service

Out of Scope

- Authentication, Discovery, Config management, Backing Service Config

Architecture



Idea & Process Flow:

- A Pricing Service whose role is to act as a master of Netflix Pricing API. APIs to view and modify Pricing information is its core feature; When prices are managed, it is expected the prices are maintained for a future date.
- As the price is effective in the future, it is the responsibility of the pricing service to notify Customer Price service of the Price change, so they could act accordingly;
- As Price change take effect, price changes must be applied to all affected customers directly(That is the problem at hand) quickly and reliably;
- This is the key responsibility of a customer price service (modelled into the same pricing service for the exercise)

- Customer pricing service will process the price change event posted, by creating another queue of customer-price change event for whom the price change should be applied
 - Note:
 - To ensure no events are lost during the process, in all the process when events are picked from the queue, temporarily a copy will be maintained in a shadow queue. On successful processing of the event, the event will be removed from the shadow queue; Redis BRPOPLPUSH is leveraged, as the operation is atomic to take items out the queue and move it to another queue. (Moved away from Redis PUB/SUB) due to potential event data loss)
 - A scavenger job is to collect events stuck in-progress queue and put it back into the main queue (Scavenger is not implemented yet) for it to be processed. By the combination of these mechanisms, no updates will be lost
- A subsequent process in the Customer Pricing Service will watch out for the customer-price event queue and update the customer price
 - Reason of updating customer price in a separate process (instead of creating a customer queue)
 - Assumption I made here is that in the real situations, customer prices may have other aspects (e.g. discounts, partner margin, extended free trial period) that needs to be considered and that may take a while at each customer level.
 - Having flexibility as above will let us scale customer – price processing by adding more nodes instead of doing the processing sequentially

Backing Services & its rationale:

- Redis is used as a persistent store for customer data; This very well can be any other storages; given the total size of the dataset, having the data in-memory is advantages for various look-ups and process
- Redis is used for Queue: Kafka is another potential candidate, But for the pricing use-case Redis Queues are perfect, when backed by the persistence, considering the limited number of events & less infrastructure components needed compared to Kafka (e.g. zookeeper)
- MySQL as the datastore for Pricing; As the data is relational went with MySQL, assuming more related objects will be stored in the future. If we prefer to unify the technology we could go to Redis (just another shard, without storing the info)


Schedule job:

- A scheduled job in the pricing service watches for price changes that should be applied and maintains various status (e.g. which price is active, which price is pushed for processing etc..)

Production read end-points:

- Spring actuator is enabled for essential monitoring of the app. Basic monitoring of the health of the status is available. Additional health checks such as the size of the queue etc can be monitored and used for altering to handle unexpected situations
- Spring health data is exposed in 9000 port, different from the application port (8080), to protect the end-point not exposed inadvertently

Note on APIs


swagger

default (/swagger) ▾

Explore

Netflix Pricing Service API

API for Accessing Netflix Pricing

Customers API : Not the main api - used for dummy customer population and verification

	Show/Hide	List Operations	Expand Operations
GET /customers			getAllCustomersOrByCountryCode
POST /customers			createDummyCustomersForS1PS
POST /customers/{COUNTRY_CODE}/{SERVICE_PLAN}/{PRICE}/{CURRENCY}/{COUNT}			createDummyCustomersForCountryServicePlan
GET /customers/{ID}			getCustomerById

Netflix Pricing Customers API : Main REST API for accessing & rollout out Price

	Show/Hide	List Operations	Expand Operations
DELETE /prices			deletePriceChangeEntryIfNotActive
GET /prices			getAllServicePlanPricesForAllCountries
POST /prices			updatePrices
GET /prices/{COUNTRY}			getServicePlanPriceByCountry
GET /prices/{COUNTRY}/{PLAN}			getServicePlanPriceByCountryAndServicePlan

[BASE URL: /]

Others

Others

- Same Redis instance is used for event queues and persistence. This must be kept in different instance/shards
- Future date validation is disabled for demo/tests. But can be enabled by commenting one line.

Out-of-scope

- Security/Authentication/JWT
- Code coverage with Unit/Integration tests
- Redis runs in a single node, cluster setup & related config
- Schema creation is left as automatic, instead of controlled schema.sql

Test-Cases:

- Key item I didn't get to are the Unit & Integration tests
- Various API Test Case for Input validation
- Incorrect Event data handling
- Killing all services as they process data and restarting them