Take Home project

Implementation Description/Documentation

Business Requirements Summary

In this assignment the requirement states that we have data store containing instrument price data, updated by external price vendors and used by downstream systems, presumably internal to the bank.

Technical Architecture Overview

The application is developed using the Spring framework and more precisely Spring Boot which allows the creation of self-contained remotely accessible applications when combined with REST services.

I have divided the program in 3 sections:

- A REST service allowing vendors to add price information to the data store
- A REST service allowing downstream systems to query the store
- The main application, accessed through the REST services and managing the price data

I have chosen REST because of it's simplicity and adequacy to the stated requirements of an externally accessible API.
Having 2 different REST services makes it easy to separate the side of the application used by external parties to add new pricing information from the side allowing to query the data and vice-versa.
These applications would be deployed to different servers/containers with a different set of security access rules.

Assumptions and limitations

Project Organisation

As it is written, the code is grouped in a single project/application.
This is done to make things easier given the time constraints of this exercise.
In a real world situation we would have 2 applications deployed as 2 jars, one to add data and the other to consume it.

On a development level this would be achieved by having 2 separate projects each one providing a public interface and making use of a common core component (in a third project) where the cache proper would reside.

Security

As written, the code does not perform any security checks or denial of service attack arbitration.
We can assume these would be provided by the bank infrastructure with a (yet to write) integration to the program code.
For instance, in the current version of the code the client accessing the API to add new price information must supply a Vendor Id. In reality such Id could be derived from the credentials supplied to access the API.
In addition, in a real world project functions calls should be validated through token to prevent unauthorised access.

Services

PriceRepositoryService

This is the service used to save and retrieve instrument prices as well as persist them to permanent storage.
This service doe not actually perform these two tasks itself but delegate them to the InstrumentPriceCache and PricePersistenceService.

This service is accessed remotely through the InstrumentPriceFeedService and InstrumentPriceQueryService facade services via a REST API.

InstrumentPriceCache

This class is the one storing Instrument Price Information for later retrieval.
It uses a strategy pattern to decide how to retrieve prices either by Vendor or Instrument Id .
The use of interface and the way this service is implemented makes it easy to replace the way data is stored. For instance both a database table or an ActivePivot store could be used instead.

PricePersistenceService

This service would provide a persistent storage to the application.
The present implementation is just a dummy one.
In practice this could be replaced by a service saving to and retrieving data from a data lake.

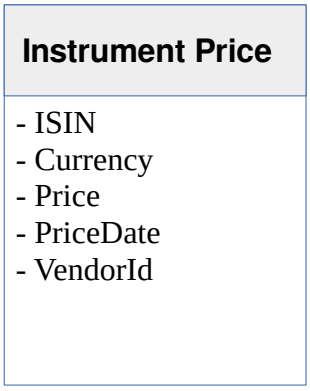
InstrumentPriceFeedService and InstrumentPriceQueryService

These services are facades giving access to the PriceRepositoryService, and are used by their respective Rest Controller to enable remote access.
In addition the InstrumentPriceQueryService also exposes a method that can be called on a schedule to delete data older than the detention retention period.
Both services use an adapter pattern to translate between public externally accessible classes and the one used in the cache implementation itself.

Domain Model

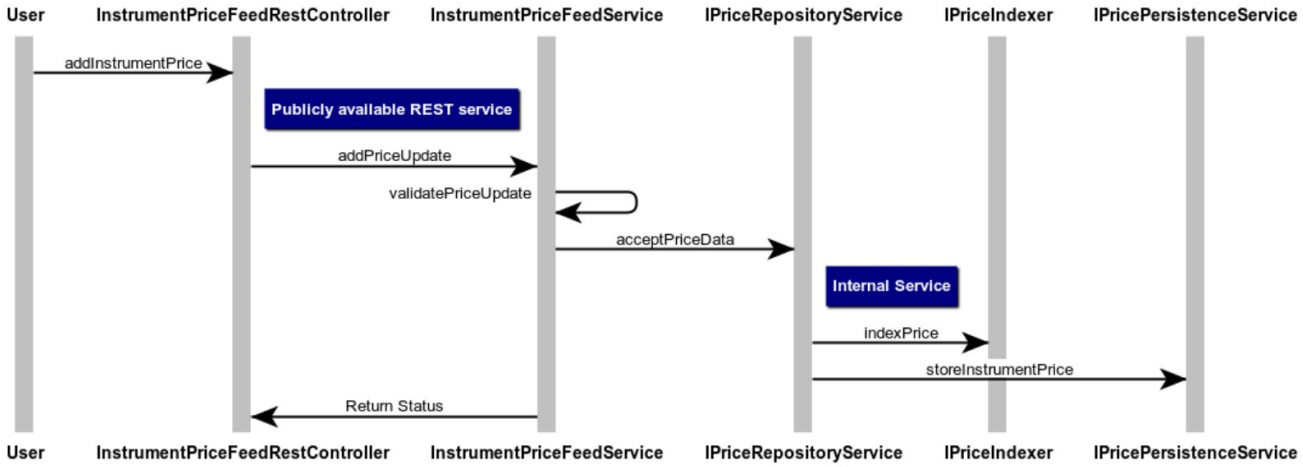
I’ve chosen to implement a very simple and flat model for instrument prices.

My choice comes from a background in agile development where the ethos it to implement the simplest solution fulfilling the requirements and then eventually propose a more tailored or complex solutions through a process of iterative development.

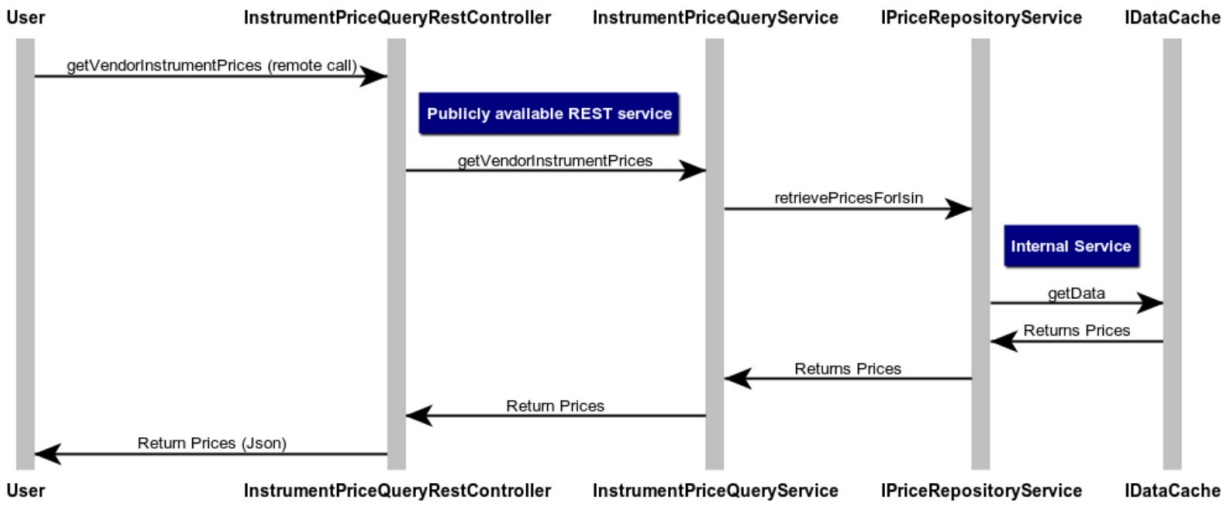


Sequence Diagrams

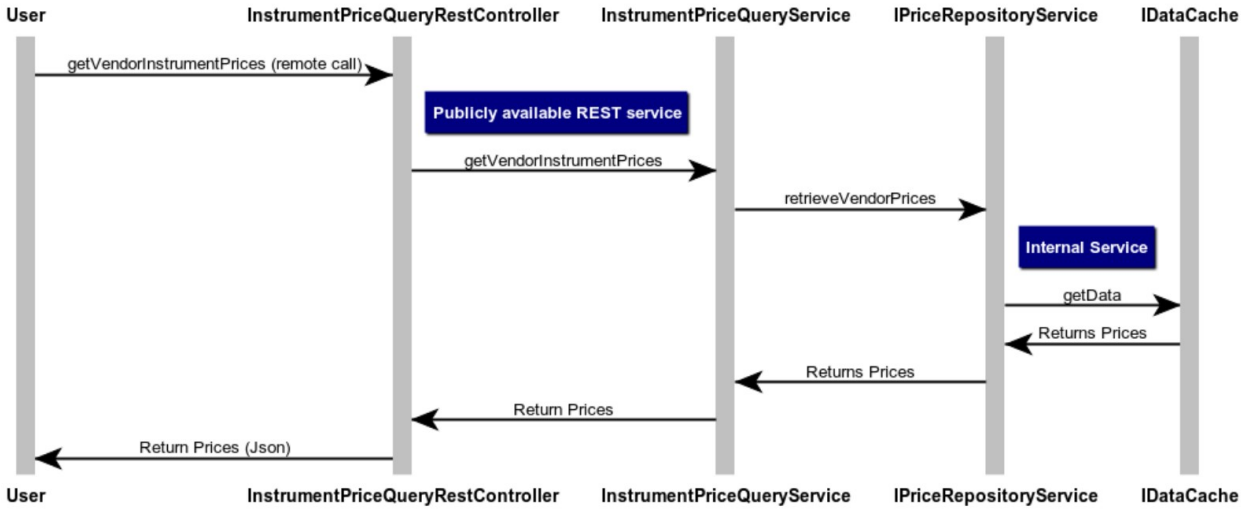
Add Instrument Price



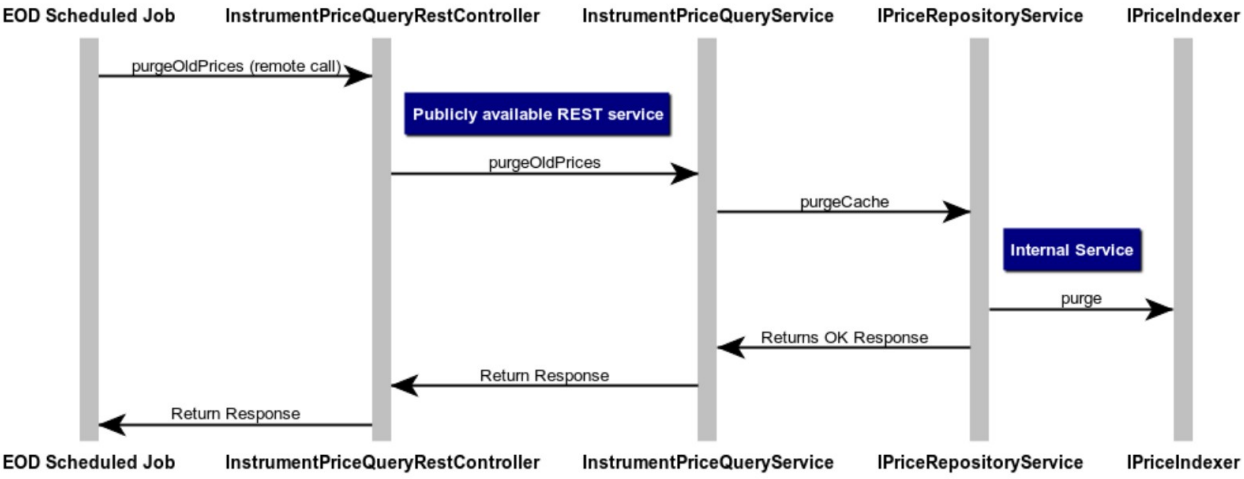
Get Instrument Prices by Instrument Id



Get Instrument Prices by Vendor Id



Delete Old Prices



Possible improvements

Scalability

The present implementation will provide a good level of service under a reasonable load, however several avenues of improvement are available should the workload increase such as:

- Enable price updates to be batched:
  Instead of having to make one call per price update, we could add the possibility for vendors to call the API to update many prices at once. The code could then be made multithreaded to give back the hand to the caller before the update has fully been processed.
- Horizontal distribution:
  We could have several instances of the services accessible through a load-balancer.
  In this situation the spread of prices updates could be spread over multiples instances of the service.
  Of course this would result in not all services containing all the data. To mitigate this, data retrieval queries would need to be performed on all instances of the servers with the replies merged to provide a complete answer.
  Another solution could also be to implement a synchronisation mechanism between the various cache instances to synchronise each others.
- Messaging system:
  A better way to improve scalability on the price update side of things would be to replace direct REST services calls by a messaging/queuing service. This could for instance be ActiveMQ or more appropriately Apache Kafka.
  Using Kafke would enable the application to scale enough to cope with peak activities when many price updates are received from many sources at once while leaving enough time to the application to process them without blocking incoming calls.
  This would also prevent data loss should the application crash.

Data distribution

In the present implementation it is up to the downstream systems to poll for new data at regular intervals or whenever they need it.

- We could implement a publisher/subscriber model where interested parties could register with the service in order to get notifications when new prices are received.
- We could also use a push model where new prices are pushed to downstream systems either whenever a new price is received or at regular interval.

Data Model

Instead of a flat data model, we could also imagine a hierarchical model with a header containing the instrument id and vendor with a collection of price/date pairs.