Mizuho Middleware Candidate Project

Author: Kevin Scully

Date: 1 March 2016

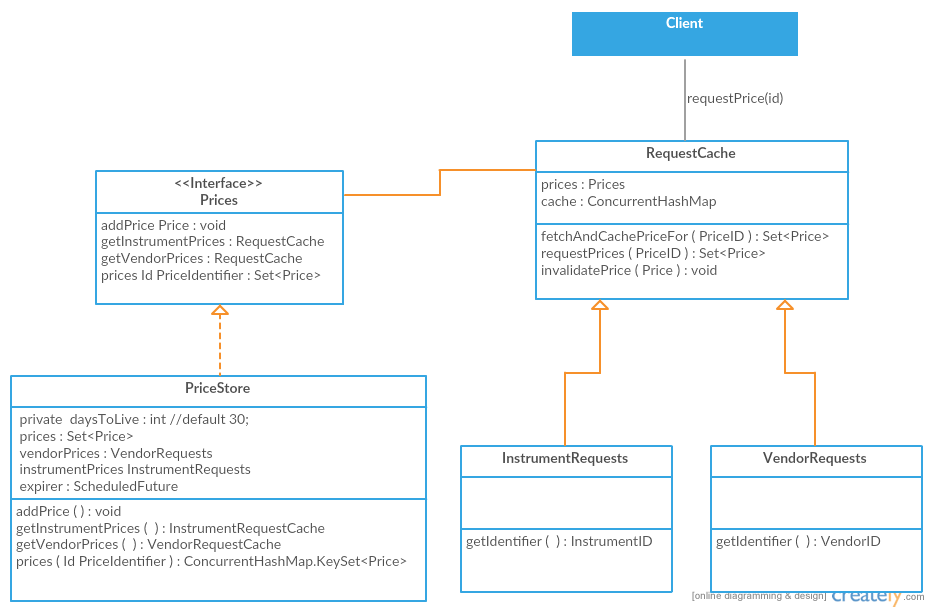
Discussion of submission for Mizuho Middleware Candidate Project

# Context

The project asks for a sample Java solution to a requirement to develop an in memory price cache.

# Core Elements

The basic elements making up the design are represented in the following class diagram.



The PriceStore manages the full list of received Prices. Prices are published to it and for this example stored in a CuncurrentHasSet (represented by a ConcurrentHashMap keyset). Prices are read from it via Request Caches, which cache particular requests, i.e. all prices from a Vendor or for an Instrument.

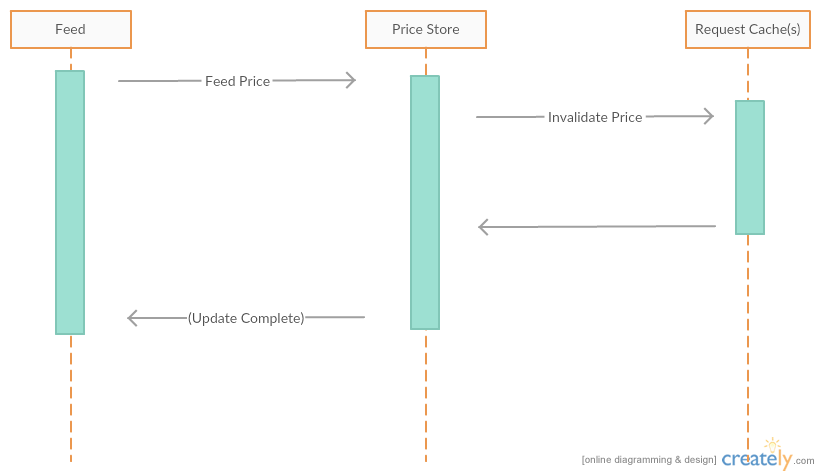
The Set of Prices could easily be persisted by a variety of means for example using the DAO pattern.

# Processing

There are a number of interactions with the Cache. The key ones documented here are Add (publishing) a price and requesting a set of prices.

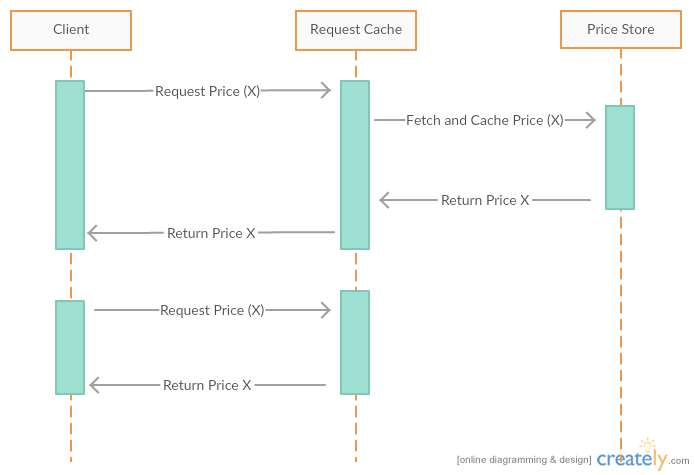
## Publishing

This represents the basic sequence for adding a price to the store.



## Request

Basic sequence for requesting a price.



# Notes

## Patterns

There are a few Pattern examples

### Singleton

PriceStore is an example of this. It’s implemented using Java’s enum feature.

### Cache

The RequestCaches are examples of a fairly standard application of the Caching pattern

### Immutable Class

Price is an example of the immutable class pattern. Initialized via constructor only, no setters, and attributes are private and final. There is no need to implement clone, as attributes (Including Instant) are immutable.

## Thread Safety

Thread safety is handled by using ConcurrentHashMap and ConcurrentHashMap keyset to represent Maps and Sets where access from multiple threads is expected. Unlike Hashtable these implementations are not locked for reading and scale much better for performance.

## Price History

On reflection I decided not to worry about keeping a price history. The requirements don’t specify or even hint at the functionality. Questioning the requirement also did not really clarify the stakeholder’s intent.

## Publishing

I questioned what was meant by “publishing”. Taking the response to questioning the requirements into account and rereading the requirements it was clear that publishing referred to the action of the feed adding (or publishing) prices to the store.

## Missing Prices

The price store as a whole isn’t really a classic cache, for one it is missing a mechanism for handing cache misses by fetching the requested data from the permanent store. Given this we handle requests for vendors or instruments not stored by returning an empty set.

## Price Equality

I decided to make prices unique based on vendor and instrument ids. This implied overriding equals and hashCode. Given this using a set to store the prices made sense.

## RequestCaches

The RequestCaches are strictly speaking, by the letter of the task, not necessary. It would suffice to search the set of Prices on each request as the set is in memory.

Given the goal to ultimately persist the set of prices though I decided in memory caches of request results might be useful. The implementation as is sets separate caches for vendor requests and instrument caches and each cache knows how to search the price store for its results. The cached results are invalidated whenever a price for a cached id is replaced in the store.

I would have been keen to abstract this further so that the searched aspect of a price being searched on (e.g. vendor) could be configured in a cache rather than using inheritance. However for this task it added no real value.