*PriceCache* Pricing Data Distribution Service  
Technical Design

|  |  |
| --- | --- |
| Author | Matthew Carle |
| Date | 13 June 2013 |
| Revision | 1.0 |

Contents

1 Introduction 1

2 Requirements Analysis 2

2.1 Instrument Identifiers 2

2.2 Historic Pricing Data 2

2.3 Client Interface 3

2.3.1 Client Type: Vendor vs. Requester 3

2.3.2 Request Type: ‘Prices for All Instruments from a Vendor’ vs. ‘Prices from All Vendors for an Instrument’ 3

2.3.3 Data Type: Latest vs. Historical 3

2.3.4 Delivery Type: Synchronous vs. Asynchronous 4

2.3.5 Putting it All Together 4

2.4 Subscriptions 4

3 Client API 5

3.1 Java Interfaces 5

3.1.1 PriceCachePublisher 5

3.1.2 PriceCacheRequesterSync 5

3.1.3 PriceCacheRequesterAsync 5

3.2 JMS Interface 6

3.2.1 Message Header 6

3.2.2 Message Body 6

3.2.3 Other Message Formats (XML, JSON, etc.) 6

3.2.4 Requests via JMS 6

4 System Architecture 7

4.1 Architecture Diagram 1

5 Technical Implementation 1

5.1 Technology Choices 1

5.2 Project Structure 1

6 Domain Model 2

7 Outstanding Issues 3

8 Running the *PriceCache* Service 4

# Introduction

This document details the technical design for *PriceCache*, a pricing data distribution service. The functionality of the system is based on the following brief received from the client:

*A number of vendors provide price information for traded instruments; a traded instrument will have different prices from different vendors. Price data from each vendor is to be cached in a local data store and then distributed to interested downstream systems. The cache will have services to allow clients to publish and retrieve data from the store. Clients are interested in getting all prices from a particular vendor or prices for a single instrument from various vendors. Prices older than 30 days must be removed from the cache.  
  
The task is to design and implement such service. Your solution must include the following:*

1. *documenting the design using sequencing diagrams, explaining choices made to arrive at the solution*
2. *documenting the domain model for persisted objects using appropriate diagrams*
3. *documenting the implementation explaining any design patterns, choice of ORM, and table design*
4. *all implemented classes must be unit tested*
5. *documenting any problems encountered and how they were overcame*
6. *Optional: provide a demo for implemented API*

# Requirements Analysis

Various aspects of the above brief required clarification. These points were discussed with the client (via email) to establish a clearer picture of the requirements. Some key points to mention are:

## Instrument Identifiers

The following additional information was provided regarding identifiers:

*“Each instrument can have a number of identifiers, vendors can use more than one identifier per instrument: either the industry standard ones (ISIN, CUSIP, etc) or their own vendor specific identifier. The same instrument is identified by the same value for a particular identifier type in all vendors of course.”  
  
“Assume that we start with an empty database/cache. The first vendor publishing the instrument and pricing information will start populating the cache and next vendors will add to it. In other words, we do not have a pre-defined list of instruments to start with.”*

This has a number of implications, as follows:

* **Matching logic:** when a new price is received, the system will try to match the instrument details on the price message against an instrument already in the cache. Based on the statement above, two instruments match where they have at least one identifier of the same type that matches.
* **Merging identifiers:** As new price messages are received from different vendors, the system will need to “merge” identifiers to build up a complete set. For example, if we receive a price from Vendor A with identifiers [RIC=IBM.N, TICKER=IBM], then we receive a price from Vendor B with identifiers [RIC=IBM.N, BBGID=IBM UN] then after both prices have been processed, all three identifiers (RIC, TICKER and BBGID) should be stored against the instrument.
* **Risk of match failure:** it is possible that a price for the same instrument is received from two different vendors, but each one uses a different, mutually exclusive set of identifiers (e.g. Vendor A specifies RIC and SEDOL, Vendor B specifies CUSIP and ISIN. In this case, there is no way for the system to determine that these two prices are for the same instrument, so it would create two instrument records.

## Historic Pricing Data

The initial requirement referred to removing from the cache any prices older than 30 days. However, it was not clear whether this implied a requirement to maintain historical prices, or whether the cache should only store the latest price for each instrument. The following feedback was received from the client:

*“As for the 30 days rule, yes correct, we’d like to purge the stale prices (older than 30 days) and I would say the main functionality should provide the latest prices, but getting the historic price for a data would be a good addition.”*

Further to this, the design presented here does cater for historic prices. However, the retrieval of the latest prices for instruments was stated as the primary requirement. To facilitate efficient retrieval of the latest prices, an IS\_LATEST column has been added to the price table (without this, an inefficient correlated sub-query to fetch the latest prices).

One additional point raised was to consider separating historic and latest prices into two separate tables. This would be worth considering as a future enhancement, although the downside is that requests to return both latest and historical prices would require a UNION in the database query.

A second additional point related to the date and time stored against prices. In the original requirement, it was not clear what the frequency of prices would be. For example, does the system store a single price per business day (i.e. a close price), or does it store intra-day prices. For the former case, it makes sense to have a separate date column, e.g. VALUE\_DATE, but for the latter, a field that stores the date and time is required. Therefore, the design includes a CREATE\_TIME column on the PRICE table.

## Client Interface

Based on clarifications received, the client interface requirements are understood as follows:

### Client Type: Vendor vs. Requester

The system has two different types of ‘client’: *vendors* and *requesters*. A client can be both a vendor and a requester.

* **Vendors:** Vendors will *publish* prices to the cache service.
* **Requesters:** Requesters can explicitly *request* prices that are currently available in the cache, and/or they can *subscribe* to the cache service to receive real-time updates for new prices.

### Request Type: ‘Prices for All Instruments from a Vendor’ vs. ‘Prices from All Vendors for an Instrument’

The two main use cases for requesters – taken from the original requirement were:

* Get prices for all instruments from a particular vendor
* Get prices from all vendors for a particular instrument.

### Data Type: Latest vs. Historical

As mentioned in section 2.2 above, the cache supports historic price data as well as latest prices for each instrument. Therefore the client interface will include methods for historic data (based on a start date-time/end date-time range) as well as methods for requesting the latest prices only.

### Delivery Type: Synchronous vs. Asynchronous

The cache is designed to support both *synchronous* and *asynchronous* delivery of data. Clearly, asynchronous delivery is particularly relevant for clients who want to receive real-time updates of new prices.

However, it also makes sense for requests for existing data in the cache. For example, if a client makes a request for all latest prices from Reuters, he can choose whether to receive the response synchronously (e.g. via HTTP) or asynchronously (e.g. via JMS). Therefore, the service is designed to handle both cases.

### Putting it All Together

The previous sections have identified four different aspects of the client interface:

* Client Type
* Request Type
* Data Type
* Delivery Type

These four aspects are the basis for the design of the API such that any meaningful combination of the four will be supported as an API method.

Please see section 3 below for full details of the API methods supported.

## Subscriptions

One of the clarifications received was that the cache service does not know who its subscribers are. In other words, the requirement is for the system to follow the *Observer* design pattern. In practice, this means that the cache service is required to publish messages to a message bus, and clients will subscribe using a *selector* specifying the criteria of the messages that they wish to receive, e.g. RIC=’IBM.N’.

# Client API

As discussed in section 2.3.5 above, the API is designed to cater for various different scenarios based on the following aspects: client type (vendor or requester), data type (historical vs. latest), request type (by vendor or by instrument) and delivery type (asynchronous vs. synchronous).

## Java Interfaces

With this in mind, the Price Cache provides the following interfaces for Java clients to use:

### PriceCachePublisher

This API is for clients to publish prices to the PriceCache. Two methods are available: one to supply a Price object and the other which takes the same information, but in separate arguments.

**public** **interface** PriceCachePublisher {

**void** addPrice(Price price) **throws** InvalidPriceException;

**void** addPrice(DateTime createTime, BigDecimal price, String vendorName,  
 Map<IdType, String> instrumentIds);

}

### PriceCacheRequesterSync

This API is for clients to request data from the PriceCache synchronously. All the methods in this class execute synchronously and return a list of prices to the caller.

**public** **interface** PriceCacheRequesterSync {

List<Price> getLatestPricesForAllInstrumentsFromVendor(String vendorName);

List<Price> getLatestPricesForInstrumentFromAllVendors(IdType idType,   
 String idValue);

List<Price> getHistoricPricesForAllInstrumentsFromVendor(String vendorName,  
 LocalDate fromDate, LocalDate toDate);

List<Price> getHistoricPricesForInstrument(String vendorName, LocalDate   
 fromDate, LocalDate toDate);

}

### PriceCacheRequesterAsync

This API is for clients to request data from the PriceCache asynchronously. Usage of this class assumes that the client has established a subscription to the service for the same instrument/vendors that they are requesting via the method call.

**public** **interface** PriceCacheRequesterAsync {

**void** requestLatestPricesForAllInstrumentsFromVendor(String vendorName);

**void** requestLatestPricesForInstrumentFromAllVendors(IdType idType, String  
 idValue);

**void** requestHistoricPricesForAllInstrumentsFromVendor(String vendorName,  
 LocalDate fromDate, LocalDate toDate);

**void** requestHistoricPricesForInstrument(String vendorName, LocalDate fromDate,  
 LocalDate toDate);

}

## JMS Interface

Requester clients who wish to receive real-time price updates, or who want to use the PriceCacheRequesterAsync interface can use the JMS interface.

The PriceCache service publishes all price messages (either explicitly requested or newly received) to a JMS topic hosted on ActiveMQ. Clients can then subscribe to that topic using a JMS selector:

* **Example selector for instruments:** ric in (‘IBM.N’, ‘MSFT.O’, ‘YHOO.O’)
* **Example selector for vendor:** vendorName in (‘Reuters’)

### Message Header

The following fields are present on the message header and can be included in selector criteria:

|  |  |
| --- | --- |
| **Header field** | **Description** |
| vendorName | Name of the vendor, e.g. S&P |
| bbgid | Bloomberg identifier |
| cusip | CUSIP |
| isin | ISIN |
| ric | Reuters code |
| sedol | SEDOL |
| ticker | Ticker |

### Message Body

The body of the message is a serialized Java *Price* object (see Domain Model section). Java developers wishing to use the JMS interface would be required to add the **price-model** artifact as a dependency in their projects.

### Other Message Formats (XML, JSON, etc.)

While a serialized Java object may be acceptable for Java clients who are happy to use the same domain model objects as the PriceCache service itself, it is not suitable for non-Java clients (e.g. C++ or .Net). Also, other Java systems may not want to introduce a dependency on the PriceCache’s internal domain model objects.

Therefore one of the suggested areas for further development would be to provide alternative message formats , e.g. XML or JSON.

### Requests via JMS

Some clients may also want to send requests (as well as receive results) via JMS. Therefore an area for future development would be to build an adapter that takes JMS messages and invokes the relevant methods on the Java API.

# System Architecture

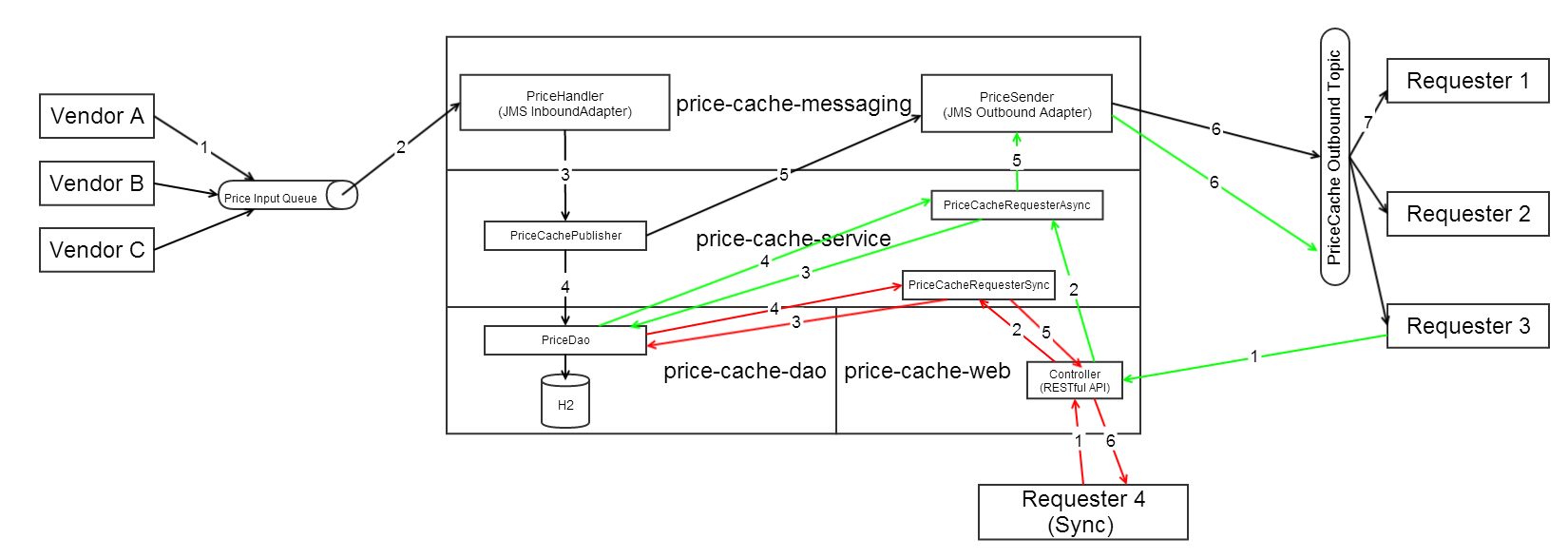
The diagram on the following page shows the high-level architecture of the PriceCache system. Key points to note:

* Vendors publish price messages via a **queue** whereas requesters receive messages via a **topic**. Prices are received on a queue because we want to ensure that each price message is processed by the Price Cache only *once* (if, for example, there were multiple instances of Price Cache running in a clustered environment). When messages are sent downstream to consumers, however, the message needs to be delivered to each consumer whose subscription criteria matches the message.
* Requesters 1, 2 and 3 are asynchronous clients (receive messages via JMS). Requester 4, however, does not subscribe to the JMS topic. They use the synchronous API to receive messages instead. The interactions for the synchronous client are shown by the red arrows.
* Requester 3 shows an example of a consumer using the Asynchronous Requester API. Note that the request is made via the web layer, but the response (one or more Price messages) is delivered via JMS.
* A real-time price update is shown by the black arrows. Messages arriving on the queue are processed by the Price Handler (JMS inbound adapter) which calls the PriceCachePublisher API. This in turn persists the price to the database (via PriceDao), and then distributes the message to downstream consumers via the Price Sender (JMS outbound adapter).

The core processing logic is as follows:

* Receive price
* Validate price message
* Identify instrument for price
  + If instrument does not exist then create it
* Identify vendor for price
  + If vendor does not exist then create it
* Merge instrument identifiers (see section 2.1)
* If earlier price for this instrument/vendor exists then update IS\_LATEST on that price to ‘N’.
* Insert new price into the cache with IS\_LATEST=’Y’
* Send the message downstream via the PriceSender component.

## Architecture Diagram



# Technical Implementation

## Technology Choices

The following technologies have been selected for the PriceCache implementation.

* Java 7
* Maven
* Spring 3.2
* Hibernate 4.2.1 - ORM
* H2 – embedded database
* ActiveMQ – embedded message broker
* Jetty – embedded web server (TO DO)
* Unitils and DBUnit – for testing DAO classes

Other libraries:

* Google Guava
* Apache Commons
* Joda Time

In general, these technologies have been selected as they are industry standard, best-of-breed technologies.

Note, however, that several of these components (i.e. embedded servers) have been made on the basis that the PriceCache is a proof-of-concept/demo project. The system has been designed, however, so that alternative components can be easily substituted.

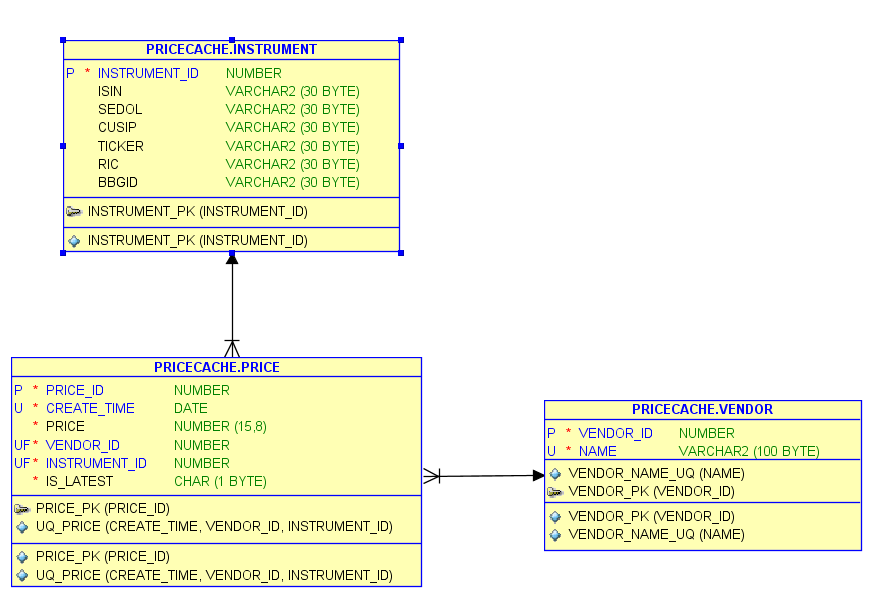
## Project Structure

The PriceCache system itself has been divided into a several loosely coupled modules:

* pricecache-parent – the Maven parent project
* pricecache-model – domain model
* pricecache-dao – data access objects
* pricecache-messaging – messaging layer
* pricecache-service – service interfaces
* pricecache-service-impl – service implementation (business logic)
* pricecache-web (TO DO) – web layer (for REST access)

# Domain Model

The domain model and the data model closely mirror one another. There are three tables/classes: Instrument, Price and Vendor. Each price contains a reference to a vendor and an instrument, as shown in the diagram below.



# Outstanding Issues

Due to time constraints, some parts of the implementation could not be completed. This includes:

* **Web layer:** Other systems may wish to interface with the Price Cache service via HTTP. For example, a RESTful interface which mirrors the functionality of and calls through to the API methods detailed in section 3 above
* **Service layer (partial):** Some of the methods in the service layer have not been implemented. However, these are mainly just different ways of querying the database and returning the results.
* **Test cases:** Test cases were completed for the DAO layer (using DBUnit and Unitils). These worked when I was running the system against an Oracle XE database, but there were problems setting this up to work with H2. Therefore these are marked as Ignored at present. Also, there was not sufficient time to complete test cases for other components. One particular area where test cases should be added is in the service layer, to test the main business logic. I would propose to use Mockito or another mocking framework here to mock the DAO and messaging layer components.
* **Remove Prices older than 30 days**: There was insufficient time to complete the implementation for this. However, it would straightforward to implement using a Quartz scheduled job running once-per-day to remove prices with a create time older than 30 days.
* **Transactions and Concurrency Control:** The DAO has transaction control implemented but further work and testing would be required to implement full distributed transactions (JMS and DB) and synchronization where necessary.
* **Enterprise Integration:** The messaging layer has been implemented using Spring JMS. This could be enhanced to use an EIP framework such as Spring Integration or Camel. This would facilitate complex event processing within the flow as well as adapters for interfacing with various different clients.

# Running the *PriceCache* Service

The dependencies for running the PriceCache service are:

* Java 7
* Maven

The system uses embedded servers for web, database and messaging, so no external set up is required. Dependencies are pulled via Maven, but note that you will need access to a repository (or Maven Central) containing the right versions of each dependent component.

To run the PriceCache service, unzip the **pricecache.zip** file to a directory on your local machine and type mvn install.

The project comes with three runnable classes for testing purposes:

* **StandaloneApp** (in pricecache-service-impl) – this is for running the PriceCache service itself, in standalone mode.
* **Subscriber** (in pricecache-client) – Sets up a subscriber to price messages published by the service. Reads criteria for subscription from system property ‘selector’.
* **Publisher** (in pricecache-client) – Used to publish messages to the service’s input queue. Prompts the user to enter details for the price message at the command line.

Unfortunately, I did not have time to configure the Maven assembly plug-in or create batch files to start these components. However, they can easily be launched from within Eclipse for testing purposes.