**Instrument Store Documentation**

*Approach*

When approaching this problem I first started by understanding what the requirements were.

As the requirement was a service which exposes a HTTP CRUD REST API I decided to use the Spring Boot web framework as I find it is the most easy to setup and also allows me to add database connectors such as Spring JDBC.

In my approach I have identified the three main Actors: **Vendors**, **Instruments** and the place which stores Prices (for which the Instruments created by different Vendors are being sold) which I called **Store**.

Given that the request was for the data to be held in memory, I opted for a H2 in memory Database rather than having some in memory cache maps, as it is more flexible, highly expandable and it would be an easy switch to a proper SQL Database.

As the information regarding the three players was limited I've decided to create the three Entities with the minimum necessary information as such **Vendor** has *id* and *name*, **Instrument** also has *id* and *name* and the **Store** has *id, price, entry date, instrument id* and *vendor id*.

I made the choice to have 3 separate entities rather than 1 (and have a vendor name and instrument name in the Store entity) as it would make the design more in line with SOLID principles and expandable in case new information was added to the Vendor or Instrument entities or if new relationships are being established with the Store entity (displayed in the DatabaseEntitiesAndDTOs.png).

As a repository I decided to use the JPA Repository as it is easily integrated with Spring, provides a lot of flexibility and also ease of use and setup.

For the time being, for those 3 Entities I created 3 Data Transfer Objects (DTOs) containing the information from the Entities which might be required for a client.

* **VendorDTO** -> *Vendor Name* and *Vendor Id*
* **InstrumentDTO** -> *Instrument Name* and *Instrument Id*
* **StoreDTO** -> *Store Id, Price, Entry Date, Vendor Name, Instrument Name*.

I used these as they make mapping the JSON object which gets sent to the client very easy.

Of course, for expandability, new DTOs can be created containing different information than the ones I mentioned.

*Design*

All things came together in order to implement a MVC Design pattern.

For this particular task, I've implemented 4 Controllers, 3 to ensure CRUD operations can be done on the Entities directly from the client side and 1 Downstream Controller which would send information to the clients which consume that API.

For the three CRUD based ones (Vendor Controller, Instrument Controller and Store Controller) I used a MVC Design Pattern (illustrated in the VendorMVCDiagram.png, InstrumentMVCDiagram.png, StoreMVCDiagram.png).

The Controllers are Rest Controllers which implement different CRUD methods, each of which has a request mapping for a different URL and with different HTTP Request Methods (GET, POST, PUT, DELETE).

In order to execute those methods, using the Dependency Inversion Design Pattern and the CQRS(Command Query Responsibility Segregation) Design Pattern I created 2 Services (a query and a command one) each of which handles different aspects of database access.

* The query does select based information and returns Data Transfer Objects as replies
* The command does insertion/update/delete executions and has void returns

These get implemented afterwards in an Implementation class which follows SOLID design pattern principles.

For the Distribution MVC Design Pattern approach (illustrated in the DistributionMVCDiagram.png) the design is a bit more streamlined as there is no need for a Command Service since we are only returning select information from our in memory database.

*Choices*

In my implementation and design I made some choices as such:

-> As it was not mentioned what type of value the price should be I went for a BigDecimal representation as it has the possibility to represent highly precise big numbers

-> I used a sequence based id auto generation for creating ids as I think it better to leave the id generation on the database side, while also having some control over how ids are going to be incremented and finding the next value

-> I used a timestamp in the database to represent the entry date

-> As the requirement for the downstream service was to bring information (either Vendor centered or Instrument centered) from the Store which was not older than 30 days, I decided to have the database handle this process, as filtering a list of objects in Java based on Dates is a much slower process, than having a sql query filter out those unwanted information.

-> I created a Utility Configuration for Date (DateUtility) which has a method that returns the current date, rather than just using new Date() as it makes unit testing a lot easier especially when checking those date based ORM calls.

*Problems and Difficulties*

Some of the choices I made, which were mentioned in the previous point, were taken in order to overcome problems and difficulties I encountered when developing the application.

-> One of them was when unit testing some of the calls which had date in them, as date varies, the ORM (repository) calls which were mocked had different millis than when the service function was tested, thus they returned no values.

-> Another Problem revolved around the representation and design of the in memory stored data structure. I decided to go for a H2 in memory Database for the numerous advantages it provided: ease of set-up, ease of generating tables, ease of initializing information (so I would have some data to work with when testing the implementation), scalability, ease of transfer to a different SQL or NOSQL based database and the speed of data access.

*Improvements*

As this is just a test case scenario the request and solution is limited and would need a lot of improvements in order to be a production ready solution.

Thus if I will mention some of the improvements I would add to this implementation.

-> Restricted access, at the time being the application can be run by anyone as there is no check to see if one is allowed or not to have access to the information displayed

-> Expose to the consumer a UUID rather than the id, by creating new columns in the tables which store this information, thus not exposing the database ids and protecting the application from possible enumeration attacks.

-> Additional information to the existing tables as well as new tables. For the time being, our store has little to no information in it, we are mentioning price as being one of the store information, but we have no idea what currency that price is in, there is no audit table to check who made changes and added or deleted information to any of our tables. Also as a good practice in case someone mistakenly removes some information there is no soft delete option in this implementation which makes it impossible to revert something in case of human error. There is also no mention of what restrictions might be on some of the tables, (can a vendor sell multiple of the same instrument at different prices <it would make sense not to> ), how long is the shop entry valid, are there any quantities of instruments which a vendor has to sell? And a lot more

-> For the time being the application works on the localhost with the default 8080 port, but going forward the application should be held somewhere either on a remote server or using some cloud services such as AWS, Azure or Google Cloud

-> Of course, the major non existing part of this application is the frontend part, which is non existing, if this application would need to be anything more than just some backend API calls it would need to have a UI to send all the calls and requests to the server side, and parse the information given from it.

-> adding a persistent database, with a distributed in-memory cache in order for multiple instances of the service get the information directly from the cache and every time the Update/Delete/Create endpoints are being called the cache gets invalidated

-> deployment is also a non existing factor, right now the application just runs locally and has no configurations regarding cloud dependency management/SDK configurations, etc.

-> Moving the application from a in Memory Database to a SQL/NOSQL Database

-> Adding metrics to provide insight into the health of the application and provide the basis for setting up alerts.

-> Creating custom response messages (for not found, forbidden, restricted, etc).