ORDER MATCHING SYSTEM

JAVIER MANTILLA

Design document

Project supervisor: Martin Hynes

GALWAY-MAYO INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND APPLIED PHYSICS

BACHELOR OF SCIENCE (HONOURS) IN COMPUTING IN SOFTWARE DEVELOPMENT

GALWAY

2018

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1. Purpose 1

1.2. Scope 1

1.3. Overview 1

2. SYSTEM REQUIREMENTS 2

2.1. Major System Capabilities 2

2.2. Major System Conditions 2

2.3. Hardware Requirements 2

2.3.1. Application Server 2

2.3.2. Database Server 2

2.4. Networking 2

2.5. Software Requirements 3

2.5.1. Application Server 3

2.5.2. Database Server 3

3. Technology Used and Why 3

4. Architecture 3

4.1. Client/Server Architecture 4

4.2. N-tier Architecture 4

4.3. REST API 5

4.4. MICROSERVICES 5

5. DESIGN Methodology 5

5.1. Functionality 5

5.1.1. Basic Use Case 6

5.2. Request and Response Format 6

5.2.1. Request Format 6

5.2.1.1. Request URL 6

5.2.1.2. Request Body 7

5.2.1.2.1. Market Request 7

5.2.1.2.2. Limit Request 7

5.2.1.2.3. Stoploss Request 8

5.3. Response Format 8

5.4. MVC Design Pattern 8

5.4.1. Model 8

5.4.1.1. User 9

5.4.1.2. PostRequest 9

5.4.1.3. Notification 9

5.4.1.4. MarketOrder 9

5.4.1.5. LimitOrder 9

5.4.1.6. StopLossOrder 10

5.4.1.7. OrderMatch 10

5.4.1.8. OrderBook 10

5.4.2. Controller 10

5.4.3. View 11

5.5. Factory Pattern 11

5.6. Microservices 11

5.7. Encapsulation 12

5.8. Inheritance 12

5.9. Polymorphism 13

6. FEATURES 13

6.1. Load driver 13

6.2. Performance 13

6.3. Featured technologies 13

7. LIMITATIONS 14

8. KNOWN BUGS 14

8.1. No automatic log refresh in UI after system restart 14

9. RECOMMENDATIONS FOR FUTURE DEVELOPMENT 14

9.1. Cancel and Update API 15

9.2. C++ code base 15

9.3. Scale-out implementation 15

9.4. HTTP/UDP 15

9.5. More design patterns. 15

10. CONCLUSIONS 15

## INTRODUCTION

The project is to create an order matching system that matches buy and sell orders for a security on a stock market. This design document presents the designs used or intended to be used in implementing the project.

### Purpose

The purpose of this document is to provide a description of the design of a order matching system fully enough to allow for software development to proceed.

### Scope

This document is for a base level system which will work as a proof of concept for the use of building a system the provides a base level of functionality to show feasibility for large scale production use. This project is focused on the base level system and critical parts of the system.

### Overview

This document is divided into 9 sections with various subsections. The sections of this document are:

* System requirements
* Technology Used and Why
* Architecture
* Desgin Methodology
* Features
* Limitations
* Known bugs
* Recommendations
* Conclusions

## SYSTEM REQUIREMENTS

The order matching system is envisioned as an integrated set of specialized modules, each corresponding to a business process, which are accessed through specific interfaces. Preliminary requirements are described in this section, first providing general requirements that apply to all modules, and then providing requirements associated with individual modules. A more detailed set of requirements will be created for each module as it is developed.

### Major System Capabilities

* System must be available on the Internet
* System must be available in market business hours per working day. Usually from 9:30 am to 4:00 pm local time.
* System must be accessible by registered users
* System must be able to log all performed transactions

### Major System Conditions

* System must use the Spring Boot Framework.
* System must communicate with clients using the JSON data format for requests and responses.
* System requests and responses must be developed on top of HTTP methods, following the REST API architectural style.

### Hardware Requirements

The following minimum requirements must be met in order for the system to operate correctly.

### Application Server

* CPU: One 64-bit core at 2.4Ghz based on x86\_64 architecture.
* RAM: 1 Gigabyte.
* Storage: 8 Gigabyte disk.
* Network interfaces: 1 Ethernet compatible card.

### Database Server

* CPU: One 64-bit core at 2.4Ghz based on x86\_64 architecture.
* RAM: 1 Gigabyte.
* Storage: 20 Gigabyte disk.
* Network interfaces: 1 Ethernet compatible card.

### Networking

* Service network: Where the application server communicates with client entities.
* Internal network: Deployed to allow communication between the application server and the database server. The database server shall not be exposed on the service network.

### Software Requirements

### Application Server

* Operating System: Ubuntu 16.04.4 LTS.
* Java Runtime: 1.8 SE Runtime Environment.

### Database Server

* Operating System: Ubuntu 16.04.4 LTS.
* Database Management System: MySQL server version 5.6

## Technology Used and Why

Selecting the appropriate technologies is critical during the inception phase of a project. Since this project adopts a microservices approach as a solution that satisfies modularity and availability, Spring Boot framework stands out in those areas at enterprise level applications. Spring Boot framework is widely used in the ICT industry.

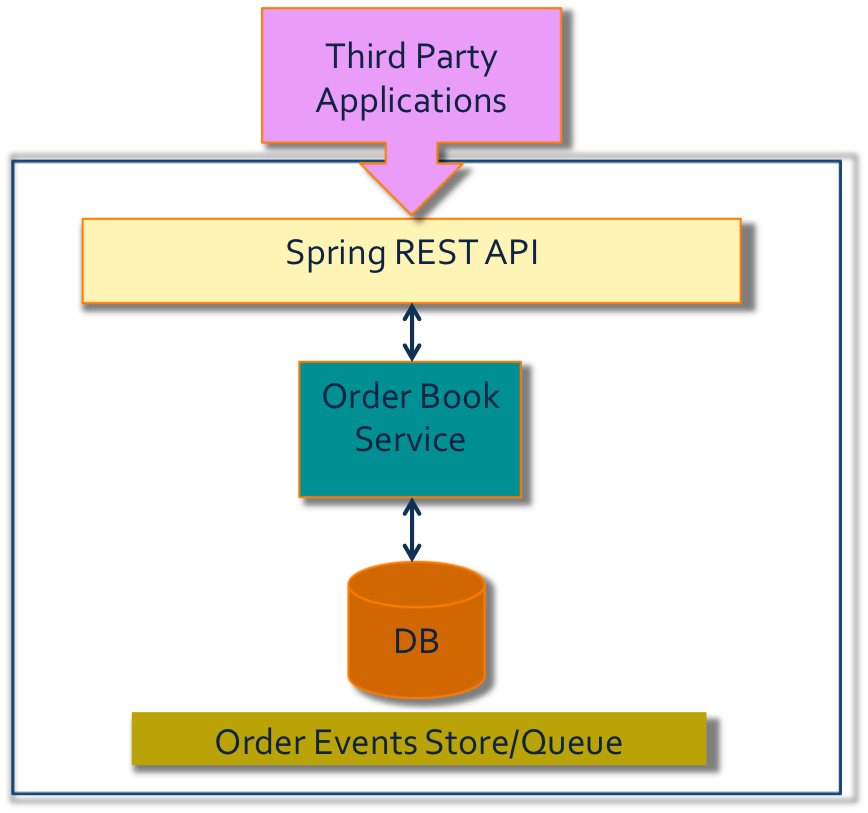
Features of Spring Boot framework in the Java ecosystem:

* Fast development.
* Mature libraries available and easier to integrate with third-party vendors.
* Convention over configuration.
* Embedded server.

Because of the already existing methods, the creation of a REST API in Spring Boot is quite straightforward. This is demonstrated implementing a CRUD style for the order matching system. MySQL is used for data persistence and the Hibernate framework is used for queries.

## Architecture

This chapter describes the environment and tools necessary to develop and execute the order matching system. The essential architectures are presented, namely, client/server, N-tier, REST API and microservices, as illustrated in the following diagram.



### Client/Server Architecture

In the order matching system, the client application and the database are separated into two parts: a front-end or client portion, and a server portion—hence the term client/server architecture**.** The client runs an application that sends orders to the order matching system following a predefine JSON request and interacts with a user through the keyboard, screen, and pointing device, such as a mouse. The order matching application receives these requests form the client application and replies to them with the respective match if proceed. The order matching system doesn’t interact with the user directly, hence is cataloged as backend software.

### N-tier Architecture

In a multitier architecture environment, an application server provides data for clients and serves as an interface between clients and database server. This architecture is particularly important because of the prevalence of Internet use.

This project has two tiers, each one containing one server. The first tier is where the application server performs, and the second tier is where the database server resides.

The advantage of this approach is better performance, better availability and better maintenance, since this design allows servers to scale out.

### REST API

REST API sits on top of the previously mentioned architectural styles.

REST (Representational State Transfer) is designed to take advantage of existing protocols. While REST can be used over nearly any protocol, it usually takes advantage of HTTP when used for Web APIs.

This freedom and flexibility inherent in REST API design allows building the API for this project. Although JSON is used for the REST implementation of this project, REST can also return XML and YAML

A REST API uses HTTP requests to GET, HEAD, PUT, POST and DELETE data. The order matching system initially uses GET, HEAD and POST methods, but can be extended to support PUT and DELETE as well.

POST method is implemented to create and post and orders into the system.

GET and HEAD methods are used to obtain information about the state of the orders in the system.

### MICROSERVICES

Like REST API, microservices architecture sits on top of the previous architectural stack. Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs.

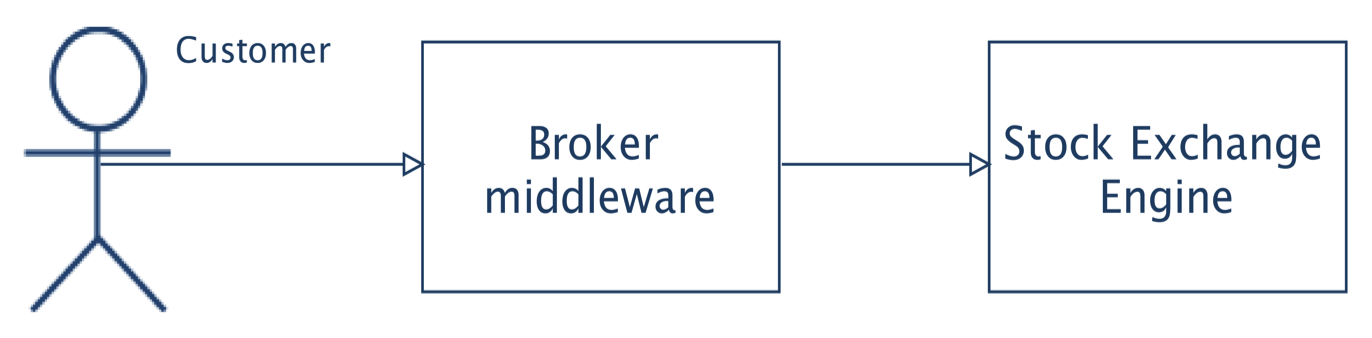
This project implements microservices for handling orders and users. This design aims for low coupling between services while maintaining a high cohesion as one application.

## DESIGN Methodology

This chapter describes how the application was conceived, design choices for its development and its functionality.

### Functionality

This project is designed to operate with 3rd party systems that in turn interact directly with end users. Those parties are mainly stockbroker firms or houses that communicate with the investor who desires to place an exchange order in the market. This order matching system is the end point where the equity exchange occurs. It is not design for end user operation, as detailed in the following diagram.



### Basic Use Case

This case begins this the assumption that the end user has already started actions with the broker firm; hence all interactions are between the broker middleware and this order matching system.

The use case starts when the broker software sends a request to the order matching system on behalf of the customer. This request is sent over a common network or the Internet. The request has a contractual format agreed by the stockbroker firm and the exchange market organization. JSON is used as data representation contained in the request. Specific market URLs and HTTP methods are used as API. HTTP is used as the medium to transport requests/responses between parties.

As received by the order matching system the request is parsed for validation or rejection. Once the request is approved, an order is created based on its contents. Then the order placed in the corresponding market, where it could be matched immediately or queued.

Regardless the final state of the order, the order matching system replies to the client system about the state of the request; if it was accepted or not, if the derived order was matched or not.

### Request and Response Format

Both, request and response are represented in JSON (JavaScript Object Notation) format. Both entities are modeled as JSON objects.

### Request Format

Requests mainly consist of two parts, an URL to reference the chosen equity or security to participate in and a body in JSON that specifies the conditions of the participation.

### Request URL

Requests URLs are of the form:

http://< IP or hostname>:<port>/order-matching-system/<stock tag>

E.g.: http://52.14.115.233:8080/order-matching-system/AAPL

### Request Body

The HTTP POST method allows requests to include more detail information about the intended market operation to achieve.

There are three requests variations according to each order condition: market, limit and stoploss. The following are the JSON objects of each one of those variations in complexity order, starting from the most basic to the most complex.

### Market Request

Is the most basic request, it expresses the desire to exchange at the current market price of the equity.

{

"userId": "dfgjkaga9", <identifies the submitter>

"stockTag": "AAPL", <identifies the chosen equity>

"type": "SELL", <action to perform on equity>

"orderCondition": "MARKET", <match at current equity price>

"volume": 10, <amount of shares to exchange>

"partialFill": true <offers for this request may or may not have to

} match equity volume>

### Limit Request

It expresses a desired price and period of time for the exchange. It has the same set of fields as market requests plus a price field and an expiration time field that sets the period of time this request is valid.

The price field accepts numbers in up to the 10000th decimal position. The minimum value is 0.0001. The expiration time follows ISO 8601 standard for timestamps and must specify a future time in respect with the current time, taking current time as the time when the request was accepted by the order matching system.

{

"userId": "dfgjkaga9",

"stockTag": "AAPL",

"type": "SELL",

"orderCondition": "LIMIT", <manifests price and time conditions>

"price": 2.5, <match at this price or better>

"volume": 10,

"partialFill": true,

"expirationTime": "2019-05-19T05:49:04.010+0000" <validity of the request>

}

### Stoploss Request

This request is a superset of the limit request. It adds an extra field for specifying a threshold price that, when matched executes the request. This is used in case of an adverse movement of the equity price in order to cut loses.

The stopPrice field follows the same convention as the price field, it accepts up to the 10000th decimal position and the minimum value is 0.0001.

{

"userId": "dfgjkaga9",

"stockTag": "AAPL",

"type": "SELL",

"orderCondition": "STOPLOSS", <sets cut-loses conditions>

"price": 2.5,

"stopPrice": 2.3, <threshold price>

"volume": 10,

"partialFill": true,

"expirationTime": "2018-05-16T05:49:04.010+0000"

}

### Response Format

The response generated by this order matching is in JSON format. It consists of a JSON object with one field called message. This field contains information about the process of the requests. The following are examples for diverse situations.

{

"message": "Request accepted: OrderId 5720d0c2-e9c7-407c-a7bb-d4626f3dbd51"

}

In this case the request was accepted and as a result an order has been spawned. The response replies with the Id of that order.

{

"message": "Request accepted: OrderId 34542f88-563e-4807-b923-06a1502d77c3 MATCHED"

}

The order spawned from the request was immediately matched.

### MVC Design Pattern

MVC stands for Model–view–controller and is used for developing this project. It divides the application into three interconnected parts.

### Model

The following classes embody the model of the application:

* User
* PostRequest
* Notification
* MarketOrder
* LimitOrder
* StopLossOrder
* OrderMatch
* OrderBook

The above express the application's behavior in terms of the problem domain, independent of the user interface.They directly manage the data, logic and rules of the order.

Each model has its persistent entity representation in the database system, except for PostRequest, which attributes are transferred to one the order family classes.

### User

This class models a registered user in the system. User entity is allowed to operate within this system.

### PostRequest

When a stockbroker JSON/HTTP request is validated an instance of this class is created to represent it in the system. This class has validation methods to verify the integrity of the request that represents.

### Notification

Responses to the client are derived from this class that collects information of the requests and order processes.

### MarketOrder

This class models the most basic order. It’s the base class for the rest of orders. It holds methods for order initialization, order equality, order placement in the specified equity market and order matching rules.

### LimitOrder

When the request specifies a price and an offer period, an object from the LimitOrder class is instantiated. It inherits fields from the MarketOrder class and appends two extra fields for price and time expiration. It also inherits MarketOrder methods and overrides some of them. It overrides the order placement method and the order matching rules, since its placement in the market is time constrained and the matching rules are now based on proposed priced expressed in the request.

### StopLossOrder

StopLossOrder class is the last child of the order family. It descends directly from LimitOrder class and extends it by one more field, the stopPrice field. This field is used to determine when to execute an exchange with the current market offer if the price movement goes in detriment of this order. Due to it’s special conditions it overrides the order matching method and order placement method.

### OrderMatch

This class models when a match between orders occurs. Its fields represent the Id of the matching orders, the timestamp of the match and how many shares were fulfilled in the transaction. Its methods are dedicated to calculate and set the fulfilled field.

### OrderBook

It is the heart and core of the application. This models the space where exchange orders for a specific equity interact. It consists of an Id for the stock tag and a set of four thread-safe maps:

* Map for sell LimitOrder
* Map for buy LimitOrder
* Map for sell StopLossOrder
* Map for buy StopLossOrder

Each map description explains what kind of order contains. There are no maps for MarketOrder objects due to its ephemeral nature. MarketOrder objects can’t match against other MarketOrder objects since they don’t have any proposed price for matching. MarketOrder objects match against prices proposed by time persistent objects like LimitOrder and StopLossOrder objects.

As maps, the key-value entries are of the form <Float, Queue<LimitOrder>>. In each map, the key is the offered price and the value is a thread-safe queue that gathers orders with that same price. The queue orders the exchange orders in a FIFO (First In – First Out) fashion. The criterion is order creation time, thus giving priority to older orders over newer ones.

The methods in this class are devoted for placing the exchange orders in their corresponding queue inside the respective map. The matchOrder method is the main player of this operation, which executes the matching logic of each class of the order family.

### Controller

This application has one controller that is in charge to attend the incoming requests via HTTP using the POST method. The OrderController class calls for the validity of the requests before its processing using Spring Boot’s @Valid annotation. It also strips the request URL to get the solicited stock tag. It replies to the client with a Notification object in JSON using the @ResponseBody annotation.

Many of the business logic that should be performed in this section is delegated to an order book service in conformity to the microservices architecture.

### View

Since this is an application that is intended to interact primarily with other computing systems and not with end users, the view is very limited and is implemented for monitoring and supervision of the system.

Data is one-way bound, from the model to the view. The view is achieved by appending to a plain text log file all the state changes performed on the model objects described in section 5.4.1.

This application uses Spring Boot’s actuator technology to expose those intern log files to a web interface.

### Factory Pattern

When a request has been validated and is accepted for processing, it is promoted to spawn an order, copying its details into the newly created order. In order to avoid code duplication, the factory pattern is employed. This also helps to avoid high coupling between client requests and exchange orders.

This design pattern can be observed in the createOrder method in the OrderBook class.

### Microservices

There are two main microservices in this application. First one is for the user entity and its pertaining operations, like registering to the system and user validation.

The second main microservice is for the OrderBook class. It is implemented in the OrderBookService class and it provides an interface to the database for order books and executes the different order book methods during the request and order processing. This is very true in its addPostOrder method.

The secondary services, MarketOrderService, LimitOrderService, StopLossOrderService and OrderMatchService, follow the data flow of controller-service-repository for the MarketOrder, LimitOrder, StopLossOrder and OrderMatch classes respectively and act as input sanitizers before the data persists in the corresponding repository.

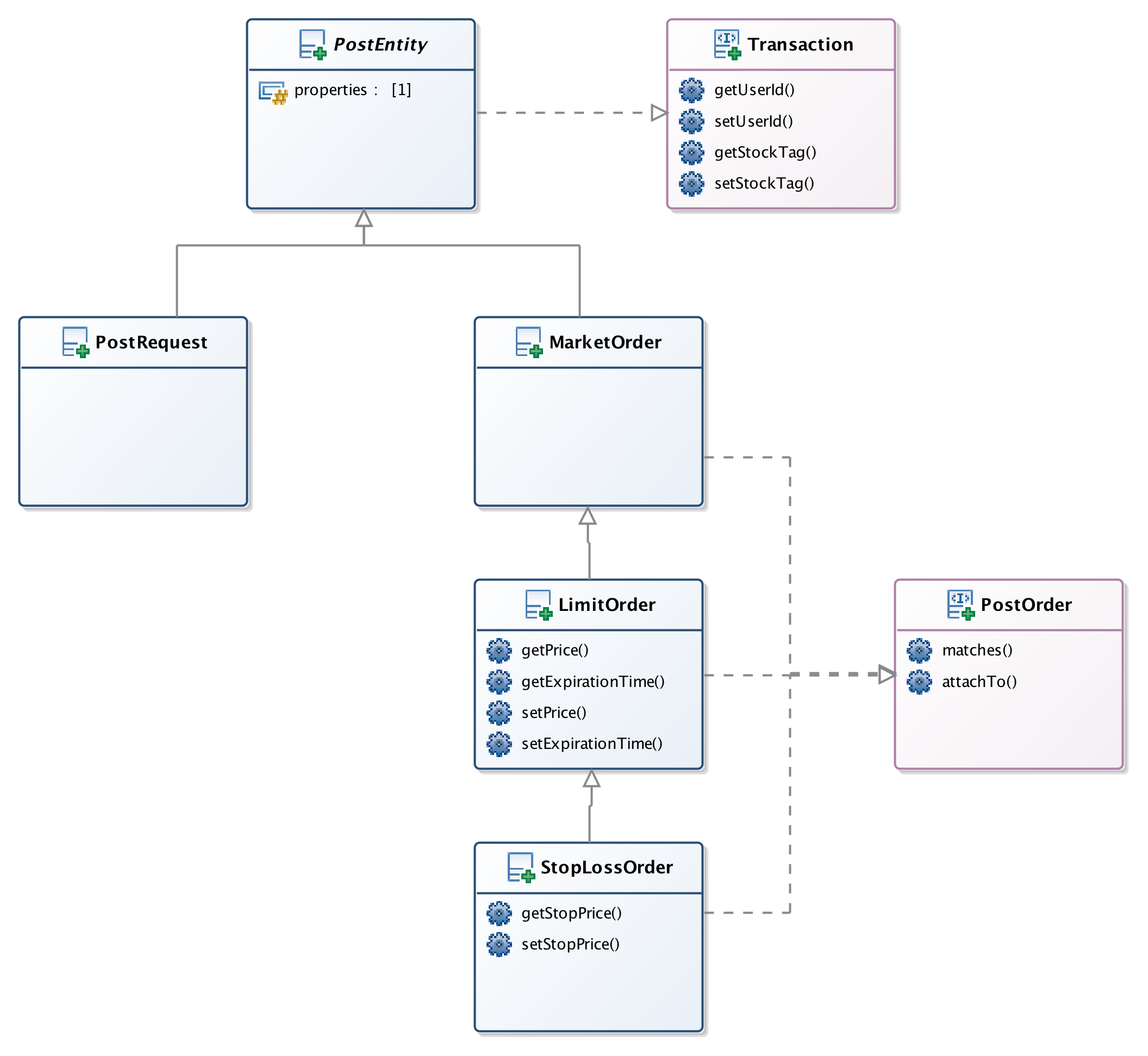
### Encapsulation

This Object Oriented Programming principle is specially applied when creating an exchange order from a valid client request. The idea “to encapsulate what varies” is implemented by copying the information from the request’s changing fields into the encapsulating order. The resulting order, in turn, has its own fields (Id, timestamp and status) independent from the ones obtained from the request.

### Inheritance

The inheritance concept is key for this project development. The order class family models customer’s exchange orders. Starting with the base class, MarketOrder, inheritance is applied in the subsequent classes, LimitOrder and StopLossOrder, each one extending the preceding class behavior.

The following diagram shows the inheritance relationships between the main model classes.



A detailed description of inheritance in the order class family is elaborated from section 5.4.1.5 to 5.4.1.6

### Polymorphism

The polymorphism principal is applied in the order class family. Each of these classes implements the PostOrder interface, as shown in the class diagram of the previous literal. This interface demands the implementation of the *attachTo* and *matches* methods. Each class has a different logic on how it matches another order and how it attaches to a equity order book, enable to use the same function call for all the classes in the order family. This avoids code duplication and allows code reuse.

## FEATURES

### Load driver

This application has a load driver in the user interface. This program loads mock requests into the application, so a preliminary visualization of the order matching system is possible.

### Performance

Thanks to the implementation of a load driver, it is possible to change its test parameters to increase the load applied to the order matching system and have an estimate of the application performance.

The system is capable to manage up to 300 requests per minute with a CPU workload in the application server of 10% and 100% in the database server. The worst response time was 1 second.

No memory leaks were detected in the performance test.

It is suspected that hibernate’s entity save operations in the MySQL server present a bottleneck in the application.

### Featured technologies

As Spring Boot framework as the main and core technology for this project, other technologies were used for different tasks where are more suitable. The following is a list of such technologies.

* Hibernate. Database abstraction and interface.
* Junit. Implementing unit tests.
* Jmeter. Implementing load driver and performance test.
* Spring Boot Actuators. Exposing internal logs to the Web UI and remote shutdown.
* Jackson. JSON parsing.
* Logback. Plain text log file creation to register model state changes.
* Jquery. Provides Ajax capability and control over Web UI behavior.
* Ajax. Link between Web UI and the application.
* HTML. User interface layout.
* CGI. Web interface for the load driver.

## LIMITATIONS

Although this application is built on top of technologies that allow scale-out software, it lacks the code implementation for it. Some of the candidate solutions for overcoming this situation are:

* Implemented as a distributed system, where objects (like order book or exchange orders) are shared across a cluster of multiple computing nodes.
* URL context load balancing. A load balancer put in between the client system and the order matching system can aid for this goal. This allows deploying a group of application servers. A load balancer distributes the incoming requests among the application server, keeping track of what request an application server is processing. This doesn’t require shared objects between the cluster nodes.

## KNOWN BUGS

After committed and corrected some issues during the development phase, a bug was disclosed after deployment is the user interface (UI).

### No automatic log refresh in UI after system restart

While restarting the Spring Boot framework during log supervision in the UI, the logs are not updated and freeze when the application comes back online.

This is due to the reset and loss of the streaming network socket that provided the log file content.

A basic workaround for this situation is to reload the UI web page, which will create a new network socket between the user’s web browser and the order matching system.

## RECOMMENDATIONS FOR FUTURE DEVELOPMENT

This project has plenty of areas where new developments can evolve. The following list highlights some of them.

### Cancel and Update API

While the HTTP POST method was modeled for order creation, the problem domain demands the capability to cancel and/or update existing orders.

### C++ code base

While the Spring Boot framework has helped immensely with this development, it puts a heavy toll on memory footprint. A new C++ implementation base will help reducing the memory footprint and enhancing performance compiling to native CPU binary code.

### Scale-out implementation

As expressed in chapter 7, this application is not developed for a scale-out deployment. Implementing such characteristic is enhances the non-functional requirement of performance.

### HTTP/UDP

Another interesting strategy is to implement HTTP over UDP, since UDP is not connection-oriented, it doesn’t create any overhead while sending and receiving requests. UDP is widely used in high throughput communications.

### More design patterns.

There are parts of the project where more design patterns can be implemented. For example, an observable pattern could be used to help to process the order matches queue into the database in a more efficient way.

## CONCLUSIONS

One of the outcomes from this project is a better comprehension of the problem domain. How the stock exchange market works was unknown for the writer of this document and application’s code. It is very important to have a clear understanding of the real domain entities in order to model them with the most

Modeling is carried out from three different perspectives. First, by modeling the domain to understand what a business does, independently of a software solution; modeling at this stage identifies the objects in the domain of the problem, their relations and their behaviors. Second, if a software solution is appropriate, modeling specifies the software objects, their relations and their behaviors; this is the software specification perspective. Third, design modeling is concerned with the distribution of responsibilities and the software control flow to achieve the required behavior.

Another gain from this application development is the opportunity and first approach in developing enterprise level applications and what it involves. Contact with enterprise tooling, like Spring Boot Framework, points out the main differences and challenges between enterprise applications and standard applications.