Time-extended, automated bidding software prototype

Nuchem Katz

Date 13/05/2021

Word count 14521 (from Abstract to Conclusions)

BSc Computing Project Report, Birkbeck College, University of London, 2021.

This report is the result of my own work except where explicitly stated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

# Abstract

A project to prototype an auto-extending auction engine is described. The methods used to implement persistence of business entities to a MySQL database and endpoints for Remote Procedure Calls for maintaining entities are included.

The project includes a range of tests that mimic the behaviour of users executing the available operations on each of the business entities. Build and environment details are included to enable the reader to replicate the tests.

The application uses Java and makes use of MySql, JSON, Spring-boot, Hibernate, Jackson and Lombok. The coding was done in Eclipse IDE with Maven used to manage the build and make third-party dependencies possible. All the project components are available on GitHub at https://github.com/nkatz01/StuProj.

In addition to implementing and testing the bidding logic for the web application, the project explores the abilities of Spring and its subset Spring-boot to provide out of the box components, configured and ready to use in a way that’s convenient and scaled and to ease dependency inversion and reduce boilerplate code. The project also explores connecting to Java with a persistence tool like MySql by the use of Jpa specification and its Hibernate implementation. Finally the project experiments with using web endpoints and the two-way transferral of Java and JSON objects via these endpoints using Jackson.

Additionally experience was gained of:

* Generalization, coding to interfaces and to separate the implementation from the specification.
* Single responsibility principle, minimizing the concerns that a given entity or operation has. To break down and disperse as much as possible the responsibilities into smaller components or operations that can be more easily traced, managed and debugged.
* Consistency:
  + the use of appropriate collections where more than one thread might be involved, the use of Optional or custom exception types where nulls or business errors might be encountered.
  + the use of database techniques such as auto-generated primary and business keys or properly engineered equals() and hashCode() to assure that persistence operations do not violate the consistency of state between objects.

# 

# Table of Contents

[**Abstract**](#_n5t1ezpib741) **2**

[**Table of Contents**](#_ckou5agwt7hk) **3**

[Glossary](#_mpbu3sx1od56) 5

[Actor list](#_6ucpw8cjjzs7) 6

[Terminology and general syntax](#_fnypkq2evblb) 6

[**Introduction**](#_kgb3caj0q1r) **7**

[Aims and Objectives](#_6uekng8yv56h) 7

[Background](#_ez6u38hup2h4) 7

[System and user requirements](#_1d7t3sniicjp) 10

[Use Case List and Descriptions](#_noj2htcis72k) 11

[**Architecture**](#_75u1dk49ms8b) **13**

[Architecture Overview](#_pej93dg8r6fq) 13

[Architectural diagram](#_68xgf7bgemha) 15

[Architectural approach](#_2vdkl3cfw4km) 16

[Initial approach](#_xqq1qulaf12d) 16

[Impedance mismatch management](#_5ryhc3pbkbsy) 16

[Persistence driven domain](#_l8jweb88hot4) 16

[Spring setting the tone](#_67l3c8skvx92) 17

[Layers complementing each other](#_gc08zemv4490) 17

[The use of Hibernate’s CrudRepository](#_1da8wmaetkim) 17

[The impact of Hibernate on Immutability](#_ot76t3yss8nl) 18

[**Installation**](#_95vqwkwsuv9p) **18**

[**Design**](#_iuwl1a882a9s) **18**

[Business logic](#_64mgwng3eahy) 18

[Non-functional requirements](#_q1r59461lso4) 20

[Decision Table (with time extension logic not included)](#_3mglgy4cgfdt) 21

[Decision table example](#_5gbsop9bpj6e) 22

[Diagram of Domain classes](#_7k362ivk45s) 22

[**Development Method and Implementation**](#_eetwhk1j0xju) **23**

[Domain classes](#_suekeneddi2u) 23

[Abstract class Storable:](#_4426h8a8xd9y) 23

[Repository interfaces](#_8r8ptxtn1dc8) 23

[Inheritance strategy](#_qulxh543n09s) 25

[Jackson annotations](#_iqnyo68x80x7) 26

[Lombok](#_5wsm7le3m0zm) 26

[Primary key strategy](#_cabxwdwys8hk) 27

[User extends Storable:](#_31t84k7j3zib) 27

[Thread safety](#_5ytovcnlngq) 28

[Json annotations](#_sj8h2lfh882j) 29

[ManyToOne associations](#_hrsjieuh5tgs) 29

[Fetch strategy](#_6v4r2tjhm1oc) 30

[Cascade strategy](#_jsz7e87jgn2) 30

[Lot extends Storable:](#_uj1kczeqz3ar) 31

[Spring dependency injection](#_iv6lqn2da6js) 32

[hashCode()](#_anf2xkm5ezys) 33

[equals()](#_s11iv9jpcu) 33

[OneToOne associations](#_yr89b6fmi8c2) 34

[Operation specification in Lot](#_pv6ih4v6q1qx) 35

[Bid extends Storable:](#_w0p8iure0lr) 38

[Controller, Service and DTO classes](#_si7w90lxjyb9) 38

[Controller, Services and DTO Class diagram](#_9y9dk7cnd1g) 39

[Endpoints for Remote Procedure Calls for CRUD Operations](#_1dvox4suh1bv) 40

[MapStruct Object Mapper](#_tr54z9i8ilnr) 40

[Encapsulation](#_71htqh138o4q) 41

[Operation Specification in Services](#_3bdcgiab3gna) 41

[**Error handling**](#_2w30f89br1m) **42**

[Custom exceptions](#_te1rjyzf8wxc) 42

[**Logging**](#_69plnq5ho5f) **43**

[**Testing**](#_v2uztnfsoyui) **43**

[Testing Tools](#_agi65z4cjexh) 44

[All test methods](#_9c1rvs9mfq3l) 44

[Testing classes diagram](#_8trammm3t2ka) 45

[Test Results](#_dfcu1x1zhxgy) 46

[Real time testing](#_pjid5w71gi4t) 47

[Overloading now()](#_h1l76z3lm2o4) 47

[Changing System.currentTimeMillis() globally](#_vnmzvtokrkzi) 47

[Passing a clock into placeBid()](#_egh25nrgwpzb) 47

[**Conclusions**](#_kybfok8q5dm8) **48**

[Achievements](#_1bjjta2umxv) 48

[Limitations](#_xaz3vgt1pyry) 48

[Future perspectives](#_3oqa9wvwqibh) 49

[**References**](#_jk3j3bu5eczj) **50**

[**Appendices**](#_1s56szcjvi8s) **53**

[Appendix 1. Installation of the Development Environment](#_3gr02u8p87ul) 53

[Java](#_r9oiyrtk0ynd) 53

[Git](#_vadx91kcvgt3) 53

[MySQL](#_mc4ubyde7b2) 53

[Eclipse IDE](#_7dybpnu0q52j) 53

[Maven](#_lfo11gldz5qq) 54

[Eclipse configuration](#_6afyieub8euj) 54

[Color](#_stf1w5wdhg2d) 54

[Lombok](#_p1wv67ml2a4c) 55

[Import project](#_gd6pdgxzmaah) 55

[Appendix 2. Usage of CRUD web endpoints](#_wzjpwnqx8wd7) 55

[Usage](#_psqo0wt0hqz9) 56

[Get](#_qet1u4c0z7t5) 56

[Get All](#_fe7wf48pi5jk) 56

[Delete](#_4863o37qa71) 56

[Delete All](#_hoinwq2z59ij) 56

[Post](#_95mb4t1rxz6b) 56

[Put](#_5rq1kd3us2gn) 57

[Appendix 3. Tables of Unit and Integration test methods](#_ycb9j23cf6ez) 59

[Unit testing methods](#_84w05t8237c0) 59

[Integration testing methods](#_nwtirw5iw95) 62

### **Glossary**

actor: a human or system process/subsystem that interacts with the system or other process/subsystem within the system

lot/auction: an instance of a specific item on auction

user: a human who is registered with the system (i.e. has a live or stored object that represents them in the system).

bidder/seller: is a user

uid/business-id: unique id (of type UUID) assocaited with each entity (user, lot or bid) and assigned at creation of the entity.

id: a unique id that gets assigned automatically to an entity by the persistence provider the first time the entity is stored in the database.

trigger-duration: a time shortly before end-time within which, raising a bid, extends the end-time ( ‘extended-endtime’ ) by a short fixed period ‘auto-extend-duration’

auto-extend-duration: a time duration by which a lot is extended everytime someone raises a bid within the trigger duration.

extended-endtime: the extended end-time of the lot due to someone raising a bid within the trigger-duration before the current endtime.

bidding-increment: a fixed amount by which the lot price increments every time a bid is placed (regardless of the amount that was specified with the bid)

highest-bid/leading-bid/current bid: the current amount offered for the lot

leading-bidder: the bidder who currently bids the highest bid

pending-autobid/autobid/proxy bid: an amount higher than the highest bid. The system bids on behalf of this bidder, bid by bid, until the pending-autobid is reached.

reserve-price: The minimum bid value that will win an auction. This is hidden from viewers and bidders.

starting-price: The first bid must exceed this to be accepted/be a valid bid.

Note: the above terms do correspond to the identifiers/attributes in the source code albeit with slight variation due to Java’s rules and conventions of how to declare them. Later on, when discussing the actual source code, Java’s form will be used for the same terms (e.g. leadingBidder).

### **Actor list**

User: a user who currently acts as a seller in the context of a particular lot or could potentially act as one (That is, there is an object representing that user but no lots related to it yet.).

Bidder: a user who currently acts as a bidder in the context of a particular lot. (They may also be a seller in which case, they’ll be referred to as a user.).

Leading bidder: a bidder whose bid would win the item if the auction ended with no further bids being made during the trigger-duration period.

Querier: someone or some process that makes a query through one of the available CRUD web endpoints - as opposed to a ‘user’ as referred to in describing the previous three actors.

### **Terminology and general syntax**

*The general assumption is that the reader has a knowledge of Java or OO programming and design*

Within this document, unless required for clarity, when referring to a certain method that a takes n parameters e.g. aMethod(**String s**), it will only be referred to, in a shorter syntax, as aMethod() unless the parameter/s is/are important, in which case it will be referred to as aMethod(String). The return type, if any, of the form bMethod()**:boolean**, will also usually be omitted, unless required for clarity.

When referring to inheritance as it relates to OOP, the terms super and sub/subclass will be used. When discussing table relations, the terms parent and child will be used, with parent being the *One* side and child - the *Many* side, of the relationship.

When referring to the domain classes in the application, the first letter of each class name will sometimes be capital and sometimes small (if referring to an instance/actual-object of the class). However, when referring to their corresponding tables in the database, all the letters will be small, since the table names are all small. Although the tables could have been specifically declared to have an initial capital letter, the former and default approach was found helpful in avoiding confusion in places where we may be unsure whether we’re talking about the class or about its corresponding table.

*Attribute* and *field* are used interchangeably to refer to attributes or instance variables of a class.

Abbreviations

CLI: Windows command line interface

eclipse: Eclipse IDE

<someVariable>: someVariable is a placeholder - to be replaced with the actual variable, without the angular brackets.

### 

# Introduction

## Aims and Objectives

The aim of this project is the design and implementation of a generic bidding engine. The focus was to formulate a set of algorithms or programs that incorporate some sensible, functional bidding requirements and non-functional bidding constraints, according to agreed bidding logic which included automatic auction end-time extension. The main objective was to design and build this engine to a point where regression tests can be written to mimic the behaviour of an actor interacting with the bidding engine. These tests explore a range of typical user actions carried out during an active auction, including setting up a user, setting up a lot and placing a bid. Testing was therefore an important component of the project.

## Background

A number of very well-known and sophisticated on-line bidding systems exist (e.g. eBay) This project arose from a personal conversation with a friend who worked with an on-line bidding system for a car scrapping company with an online presence. It became apparent that the feature of automatic end-time extensions (auto-extend) is absent from modern and popular bidding platforms such as eBay and LiveAuctioneers[1](#kezlaw88tq8c). Auto-extend is a feature vital to in-person auctions, whereby the auction’s end-time extends automatically, so long as there is action (i.e. placement of a bid) in the last few minutes of the auction. This provided the basic motivation for the current project. Other online bidding platforms like eBid and Tarisio (fine instruments) do implement auto-extend but have different ways of doing it. Whilst some like Trasio and The Sale Room[2](#88o9qqyrplh6) have auto-extend for all auctions, eBid offers it as an optional setting when starting the auction. The period which triggers the extension and the extension time itself also differs among them in both, proportionality and length. For example, eBid[3](#2mg5a3v2ir4v) gives 60 seconds for the trigger period and 60 seconds for extension while Tarisio[4](#50foyycnuiie) has 10 minutes for the trigger period and 15 for the extension. The available online bidding platforms also differ in the way they treat regular bids vs proxy bids. While eBay requires for a proxy bid to be specified as such, Tarisio treats certain bids, those left on the phone or prior to the start of the auction, automatically as proxy bids. The decision table (Gane & Sarson) included in this project was modified from the one used for the car scrapping bidding system.

Although contributing little, two prototypes of bidding implementations in Java were examined as part of an early exploration of existing approaches. One source was on [Stackechanges](https://codereview.stackexchange.com/questions/179888/simple-online-auction-system-in-java)[5](#1wgph8duqec8) and the other on a [private individual’s](https://github.com/krishnakarki195/Bidding-System-In-Java/tree/master/src/main/java/com/bidding)[6](#doum57dvta2i) github account

The following components and aspects were included in the design and development throughout the building of the application:

* A persistence layer in line with the JPA specifications and provided by Hibernate’s implementation with MySql as the underlying databases
* A presentation layer enabled by spring-web, spring-data-rest and the spring-tomcat starter dependencies

The motivation for using Spring in this project was a short three month period job that I held as a Java junior developer in the Winter term of 2021. My experience in the company was of Spring being a very useful and necessary framework for working with Java in the Software development industry. I therefore wanted to use the project as an opportunity to gain experience in it.

The benefits expected from using Spring were the following:  
1. To provide a pseudo-microservice web application, exposing web endpoints with which outside users can interact.

2. To reduce the configurational burden and boiler plating that would be required during the process of adding a persistence provider (such as JPA and JDBC with MySql) and a web engine (such as tomcat and servlet and controller mappings) in their native form.

In general, to support dependency inversion and help write concise code, this project relies heavily on the use of autowiring and bean declaration in its various forms, and in a range of Spring annotations and classes as well as other important libraries such as Json, Jackson and JUnit.

In terms of the underlying philosophy driving the design of the application, attention was given to principals such as generalization, scalability, separation of concerns and dependency inversion.

In preparation for this project, the following material were read through and consulted to see if and how they can aid in the decision of which design methods and technologies to use for this project:

**Object oriented systems analysis and design using UML**[**7**](#4v58wqt3zr6r)(Third edition - 2009) by Simon Bennett, Steve McRobb and Ray Farmer to see if and how it can inform the planning and design of this application.

[Java Tutorialspoint](https://www.tutorialspoint.com/java/index.htm)[8](#7mpu4stp1tqn) to refresh the knowledge of the language.

[Microservice](https://www.tutorialspoint.com/microservice_architecture/index.htm)[9](#aru67csms4r1) architecture to understand what they are and why they are useful, before beginning to use Spring web service technologies.

[Eclipse](https://www.tutorialspoint.com/eclipse/index.htm)[10](#m9d4ssrgiyz7), [Maven](https://www.tutorialspoint.com/maven/index.htm)[11](#vf3uwb84n3d6), [Gradle](https://www.tutorialspoint.com/gradle/index.htm)[12](#2mphjan3ae1o) and [Spring](https://www.tutorialspoint.com/spring/index.htm)[13](#6fosjaphesy) as prerequisites to [Spring boot](https://www.tutorialspoint.com/spring_boot/index.htm)[14](#27jwxcqkbfyv) and as a result also [Groovy](https://www.tutorialspoint.com/groovy/index.htm)[15](#5zq5a7d6ajvf) as a prerequisite to Gradle. [UML](https://www.tutorialspoint.com/uml/uml_object_diagram.htm)[16](#ydj22bx5o44j), [Design patterns](https://www.tutorialspoint.com/design_pattern/index.htm)[17](#xwvk0kw4jl7y), [Jackson](https://www.tutorialspoint.com/jackson/index.htm)[18](#nhhn08sz64k3) and a [tutorial](https://www.tutorialspoint.com/restful/index.htm)[19](#wdc1ked0stx) about RESTful web services were also consulted.

Throughout building this application, the following sources were instrumental in helping to practically achieve implementing the many features and use cases that this application is made of:

* All areas and specifically core Java: [Bealdung](https://www.baeldung.com/)[20](#r4emxpnn5d8v), [dzone](https://dzone.com/)[21](#w23x7t8euxwr), [jenkov](http://tutorials.jenkov.com/)[22](#m4idfopdnnzc), [oracle](https://docs.oracle.com/en/)[23](#80vwmslvzd07), and Javadoc for whichever class/method I was developing.
* Spring framework: [Spring tutorials](https://spring.io/guides#tutorials)[24](#c0vts3ooh2px), [dineshonjava](https://www.dineshonjava.com/)[25](#fn5sk885xvym)
* Hibernate and JPA: [howtodojava](https://howtodoinjava.com/)[26](#eufkukq3at95), [vladmihalcea](https://vladmihalcea.com)[27](#4l4egy8n256l), [thorben-janssen](https://thorben-janssen.com/)[28](#vv56qgygxvig),
* JSON, Jackson and data transfer:: [stackabuse](https://stackabuse.com/)[29](#fpj4hiny8jbv), [andrewtarry](https://andrewtarry.com/)[30](#8iqg0hhlhfv) and [auth0](https://auth0.com/)[31](#btqk93da6vfb)
* Lombok: [project lombok](https://projectlombok.org/)[32](#pe5it4h48kji)
* Logging: [Logback project](http://logback.qos.ch/)[33](#jp2bhm3lv61s)
* Testing: [hamcrest](http://hamcrest.org/)[34](#q0oy19nfs4za)
* Specific questions and answers to 'brick wall' problems: [Stackoverflow](https://stackoverflow.com/)[35](#mbcqigfey21c),   
  [eclipse org](https://www.eclipse.org/)[36](#snxa0hecnwtm)

## 

## System and user requirements

1. To build the system such that it can run locally as a web service and be accessible from a web browser.
   1. To enable the running/execution of the project as a Spring-boot web application, whether from eclipse IDE or from the CLI. The various endpoints specified in the controller component of the application will then be reachable through a web browser locally, with port 8080 as the root by default.
2. To expose endpoints for the basic CRUD (Create, Read, Update, Delete) operations in a web browser.
   1. The basic CRUD operations, which are Create, Read, Update and Delete will be invokable on the above entities, at their designated endpoints, via platforms such as [Postman](https://www.postman.com/)[37](#8f555jhmtjun) or by using the traditional curl facility, so long as the application is running on the server.
      1. Note, whilst the process of creating a lot and placing a bid internally (e.g. from within the test files) must conform to a set of rules and constraints, formulated by the bidding logic algorithms, in order to succeed; currently, querying the CRUD operations for these entities via the endpoints, if in the right format, will successfully execute their respective operations without giving heed to the aforementioned bidding logic. That is, they will not enforce anything byoned the basic rules such as primary key/foreign key and ‘column not-null’ constraints, and the like, imposed by the underlying database. For any production system, this feature would have to be removed..
3. To allow the creation of a user, whether as a seller or a bidder.
   1. In order for a user to be able to create a lot or bid on one, they first need to exist as a user with the attributes such as a username, uid, the lots they created and the bids they made on other lots, recorded with them.
4. To allow the simulation of a seller creating an item (lot) to be auctioned.
   1. Once a user has been created, it can be used to simulate the creation and initiation of a lot (listing). A lot is created together with details about the item (e.g. description, bidding-increment, end-time and optionally also a reserve-price) and is then linked up with the user by giving the user object a reference to it.
5. To allow the simulation of bidder placing a bid on a lot
   1. Once a lot exists, a bidder would be created (if it doesn’t exist already) to represent the user who attempts to bid on the lot. The creation of a bid, together with its amount, would then follow to simulate the placement of a bid on the existing lot. Details associated with the bid e.g. the bidder who attempted it and the lot it is placed on, as well as a uid are recorded with the bid.
6. To allow for users, lots and bids to be persisted in a database
   1. Any of the three domain entities, a user, lot or bid, can be stored away in the database for future retrieval. Though, a bid cannot be stored if the lot it is tied to, isn’t in the database already. The same is true for a lot in relation to the user who created it. However, if a user is stored, and it has a lot related to it already, the lot will be stored automatically with the same save() operation invoked on the user. Same is true for lot -> bid.
7. For the logical behaviour exhibited throughout the process of *placing a bid* to reflect a reasonable and sensible behaviour as per the *bidding logic* section under *design*. Examples of this include:
   1. When a bid is first attempted on a new lot, the bid has to be higher or equal to the starting-price, if there is one.
   2. A bid that is lower than the current highest-bid, is refused.
   3. If a bid is put in after endtime, it is refused.

## 

## Use Case List and Descriptions

Below is a table of the above requirements in the form of use cases together with a description for each use case.

|  |  |
| --- | --- |
| **Use Case** | **Description** |
| Create User | A user can be created together with a username, possibly zero or more lots and zero or more bids (on other user’s lots) associated with them. |
| Create lot | A lot can be created together with relevant details about the lot. The relevant details are (some of them optional or have defaults): the user who lists it, zero or more bids associated with this lot, its starttime, starting-price, title, description, reserve-price, bidding-increment, trigger-duration, auto-extend-duration, time-zone, endtime. |
| Place bid | A bid can be placed on lots created by users other than the user/bidder who created this bid. A bid can be either a default or fixed, pre-set bidding-increment, or it can be for an arbitrary amount, in which case the amount is rounded down to the nearest bidding-increment. All bids are autobids in that, if they’re more than 1 bid increment higher than the current leading-bid, the system bids on the user’s behalf until the autobid amount is reached. |
| Get bid by the order they are found in the set | Sometimes it can be useful to retrieve one or more bids from the set of bids maintained within the lot, with no regard to which one specifically, we’re getting back. E.g. going from zero bids to one, and then retrieving the first bid, to see if it equals the bid that was just placed and to update it, if we wish to. It can also be useful if we want to cycle through and retrieve all the bids in turn. |
| Save, Find, Exists and Delete from Database | Those operations and more are available on each of the three domain entities in relation to their corrosponding repositories that represent their state in the underlying database - in our case MySQL. |
| Add new entity | A querier can, from Postman or curl, submit a (pseudo[38](#dxl81j5e59ah)) remote *Post* request from the ‘create’ endpoint in order to create and store any of the three domain entities into the database. The body of the query would contain a json object specifying the type of the entity (user, lot or bid) and optionally any of the attributes/properties that this entity maintains in its class and corresponding database table. Though, not-null attributes have to be provided whilst some of the others will take default values.  Objects can be nested and can be specified to relate to already existing entities. E.g. a new bid within a new lot, within a user, that exists already in the database. In cases like this, the existing entity would be identified by its primary key in the record of the table in which it is held. |
| Update entity | Similarly, a querier can submit a *Put* request from the ‘update’ endpoint in order to update any of the three domain entities. A DTO object representing that domain entity object desired to be updated and containing partial data of that domain entity would be sent through in the body. Since each DTO object has a dedicated mapper to translate it to its corresponding domain entity, each entity update requires its own separate Put request and cannot be nested within its parent, e.g. lot inside its user. |
| Get all entity records | A querier can also submit a *Get* request via the ‘allents’ endpoint to view all the records of all the entities in the database. The entities will be returned to them in JSON format and be presented in a combined form, that is, from the top most parent -> user at the outer edge of the structure to the bottom most child -> bid, at the innermost position in the hierarchy. |
| Get specific entity | A querier can submit a *Get* request at the ‘getent/{id}’ endpoint to view a single item or record in the database out of any of the three domain entities. An *id* of an existing record has to be submitted together with the request, else an ‘entity not found’ response will be relayed. |
| Delete entity | A *Del*  query to delete a specific entity can be requested from the ‘delent/{id}’ endpoint. The id of an existing record has to be provided and the record requested to be deleted cannot violate any of the basic database enforced constraints such as the request resulting in leaving orphan records in the database. E.g. requesting deletion of a user, will delete all the lots and bids related to it but a request to delete a bid that has its bidder as the leading bidder on a lot, owned by a different user, will be denied. |
| Delete all entities | A Del query to delete all entities at once from the database by hitting the ‘delents’ endpoint will result in all the tables emptying their content. |

# 

# 

# Architecture

This section is going to explain the architectural design decisions and the reasons these were chosen.

### Architecture Overview

The project is made up of different subsystems or subcomponents, realized by breaking the source code into packages so that each subcomponent only has visibility and dependency on other classes that are very closely related to them. Only those classes that need to see all, reside in the top root package of the application. Specifically, BiddingSoftwareApplication.java is what drives the application and needs access to all the sub packages in order to scan them e.g. bean registration and other Spring configurations. The sub components of the project are outlined below:

**Subcomponents**

Domain: part of the system that deals with holding and manipulating the data about lots, users and bid entities.

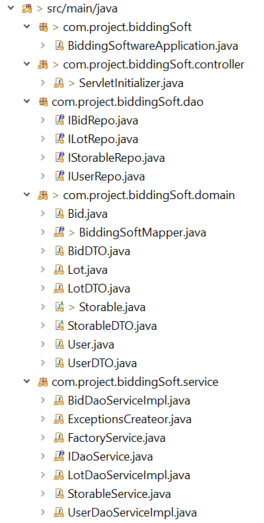
Data Transfer objects (DTOs): objects whose purpose is to expose - and allow for the modification of - parts of the domain entities/classes of the system.

Data Access Objects (DAOs): part of the system that deals with the storage and retrieval of data

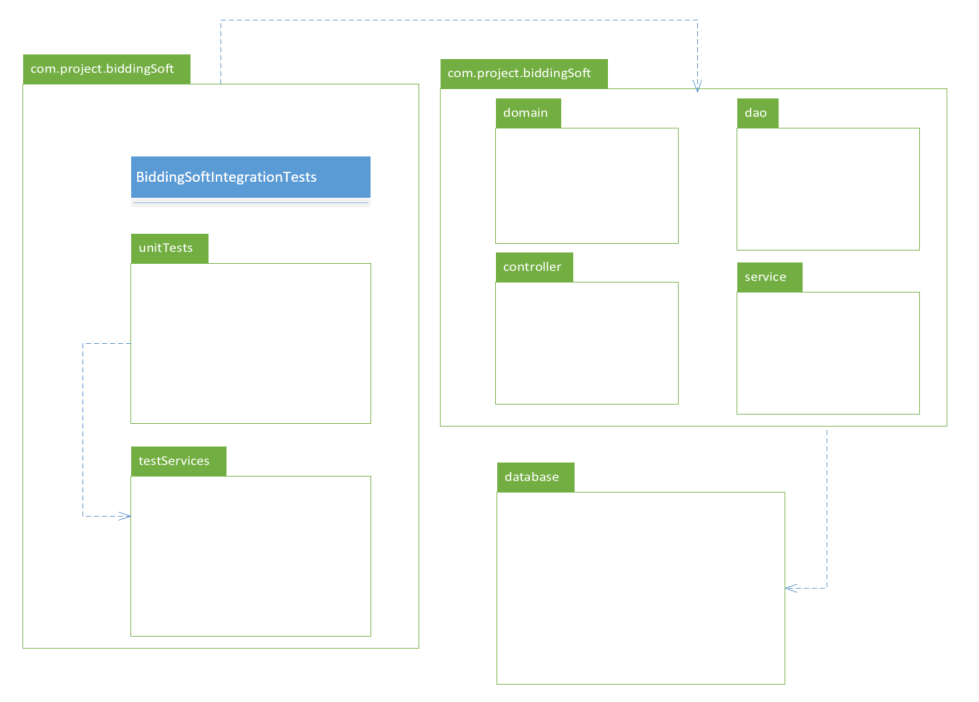
Controler: part of the system that deals with exposing the web endpoints, receiving requests and serving/displaying content or feedback to the external actors/queriers issuing those requests.

Service: part of the system that deals with controlling the interaction between the above subsystems

The following shows how the sub components are realized; by the organization of the source files within *packages* inside the src/main/java folder that is within the overall *biddingSoft project*:



### Architectural diagram

A UML system architecture diagram showing the subsystems within the application.

### Architectural approach

This section deals with the reasons behind choosing a horizontal development approach towards the three layered architecture that this application consists of, rather than a vertical one. That is, the choice to develop all three layers alongside each other rather than one at a time, in a subsequent fashion.

#### Initial approach

At the outset, the approach was to first draw up a list of requirements, use cases, their descriptions and diagrams to illustrate them. The intention was to then code the domain classes and bidding logic within their operations, test them to see that they work and only afterwards look into extending the application to include persistence and also, if time permits, an API with exposed front-end HTTP endpoints, that is: a controller or presentational component.

This approach was rejected because of the following reasons:

##### Impedance mismatch management

Initially, after drawing up and attempting to implement a list of requirements, use case descriptions and class diagrams, It became obvious that elaborate domain classes would highly complicate the implementation of persistence, due to the burden of mapping the attributes and references within the entities to the columns and relations in the database. Similarly, a too deeply nested hierarchy with a large number of relations would likewise complicate the controller and dao services when storing or updating newly created entities because of the need for manual checks of the states of all the relations (e.g. determined by fetch and cascade strategies). It was therefore necessary to implement persistence early, when the domain classes are still limited and uncertain in order to determine the limits/size of the domain classes.

##### Persistence driven domain

Another determinant to rejecting this approach was the hypothesis that ultimately, persistence should have a greater influence on how the domain classes are modelled, and as a result also on how the controller behaves in response to requests, rather than the other way around - the domain classes deciding or at least being independent of the persistence layer. (Although ideally, the two should be completely separate and decoupled, some coupelling like the necessary annotations in the domain classes to reflect the design decisions of the underlying database tables, are unavoidable. Similarly, some coupling between the controller classes and the persistence classes was also allowed as overgeneralizing hierarchies and dependencies would have led to excessive duplication of classes and operations which, for an application of this size, just for the sake of demonstrating generality and scalability, seemed unnecessary and needlessly verbose.)

It was therefore realized that introducing persistence after the classes have been modelled and written, might require entirely restructuring and re-coding the domain classes. This hypothesis turned out to be well founded as will be demonstrated later.

##### Spring setting the tone

Lastly, it was decided, from the outset, to use Spring as the framework for this project with one of the reasons being so that later, when it came to look into persistence and API end-points, Spring would much reduce boilerplate code. Moreover, using Spring would allow for the use of specifications and implementations of frameworks such as spring data-JPA, Hibernate and spring-web and much more. These frameworks provide highly sophisticated out of the box functions, shielding one from having to dig into the intricate operations being carried out under the bonnet by these frameworks. In addition, Spring’s provision for dependency autowiring, testing capabilities and for the ease of configuring the different components together were expected to be very useful.

The decision to use Spring required right from the start to have a basic surrounding Spring structure to the application. It’s also convenient to know at the outset, when generating the project skeleton, roughly what *spring-starters*, for the various layers in plan to include, would be needed throughout the application. In our case, these starters were included in the pom.xml file at the start of the project. (By basic surrounding structure we mean bean registration and management and the relevant annotations that cater for that (e.g. @Component, @Bean and @Configuration) as well as a SpringApplication.run() statement in main().) Spring *requires* for beans/components to be properly annotated and that any other dependency/technology that is incrementally added to the project is appropriately consolidated/configured, under/with Spring’s supremacy, instead of interfering with Spring. Once the basic steps to have the application start (in our case as a web service) have been followed, it’s mandatory to continuously test and be vigilant that these *requirements* are never violated. Thus going with Spring meant it would be easier to integrate the other layers right from the start so that they can be continuously tested together rather than in sequence.

#### Layers complementing each other

What follows are two examples of decisions that were taken in an individual layer of the application and were better informed as a result of developing the three layers alongside each other, which in turn was also as a result of choosing Spring as explained above.

##### The use of Hibernate’s CrudRepository

The first example is the early decision, at the start of the project, to use an extension of Hibernate’s CrudRepository as the persistence strategy and not resorting first to more lower level strategies such as the use of sessions or EntityManager. Learning that Spring takes care of low level details such as transactions, repository thread safety and session availability helped in making a wise decision early on, not to get over involved with the persistence layer.

##### The impact of Hibernate on Immutability

A second example of making an educated decision early enough in the project, this time in the business layer, based on a choice for the persistence layer was the question of immutability. Initially, a strong goal was to declare as much as possible, all attributes final and whenever an object’s attributes needs to be changed (e.g. re-setting the amount of a bid to the nearest biddingIncrement during rounding) a new copy of the object would be created with all but this attributes the same as the old. However, in a real application of this kind, a dependency such as a lot or a user cannot be randomly generated (as we did for the purpose of testing) and so would ultimately need to be provided during construction. Similarly, immutable attributes, even those where *defaults* do make sense also from the business POV, would have to be set during construction (or field initialization) and this can indeed be done with Spring’s constructor injection. Unfortunately though, JPA specification, the rules that Hibernate our persistence provider implements, requires a parameterless constructor and although not impossible, this requirement makes everything quite a bit more complicated as [this source](https://allaroundjava.com/immutable-entities-hibernate/)[39](#qqnps6qqbve9) shows. It was therefore decided, in good time, to not focus on immutability, throughout this project.

# Installation

For instructions on how to set up the development environment and related integrated tooling, please consult [appendix 1](#_95vqwkwsuv9p).

# Design

The rules that serve as the basis for the design of the bidding logic algorithm are detailed below. A summary is shown in the table “Decision Table”.

## Business logic

1. All bids are **autobids**.
2. Lots have an auction **start-time**, **end-time**, **extended-endtime**, an **auto-extend-duration** and a **trigger-duration**
3. When a lot is set up the **extended-endtime = end-time**
4. The **bidding-increment** is held on the 'lot' object.
5. The **starting-price** is held on the 'lot' object
6. Bids below the **starting-price** are invalid.
7. Bids are rounded down to the nearest **bidding increment**
8. All bids are recorded together with the ‘lot’ and ‘user/bidder’ associated with them.
9. All lots are recorded, linking up with the ‘user’ who created it and the ‘bids’ placed on it.
10. Before the auction starts the **highest-bid** = 0.0.
11. A valid bid must be at least
    1. the **starting-price**, if it’s the first bid and there is a **starting-price** set on the lot
    2. 1 **bidding increment,** if it’s the first bid **and** there is **no starting-price set on the lot**
    3. **1 bidding-increment** more than the **highest-bid** if it’s **not** the first bid on the lot
12. Following a new **bid**:
    1. If there is no **pending autobid**
       1. If this new bid is the first bid
          1. If there’s no **starting-price** set on the ‘lot’ the **highest-bid** becomes one **bidding-increment** higher than 0.0.
          2. If there is a **starting-price**, the **highest-bid** gets set to the **starting-price**, provided this new bid is higher or equals to the **starting-price.**
       2. If this new bid is not the first bid, the **highest-bid** becomes one **bidding-increment** more than the previous **highest-bid.**
       3. In all 3 cases, this new **bid** becomes the **pending-autobid** if it is one or more **bidding-increments** higher than the new **highest-bid**. The bidder of this new bid becomes the **leading-bidder**.
    2. If there is a **pending autobid**,
       1. If this new bid is less than the **pending-autobid**, the **highest-bid** becomes one increment higher than this new bid. The bidder of the **pending-autobid** remains the **leading-bidder**.
       2. If this new bid is equal to the **pending-autobid**, the **highest-bid** becomes equal to the **pending-autobid**. The bidder of the **pending-autobid** remains the **leading-bidder**. The **pending-autobid** is removed.
       3. If this new **bid** is greater than the **pending-autobid**, the **highest-bid** becomes one **bidding-increment** more than the **pending-autobid.** The bidder of this new **bid** becomes the **leading-bidder**. If this new bid is higher than the new **highest-bid**, it becomes the **pending-autobid** else the **pending-autobid** is removed.
13. END OF AUCTION
    1. The auction ends when the **extended-endtime** is reached**.** No bids are accepted after this time.
    2. If a bid is placed within the period **extended-endtime - trigger-duration** (i.e. before the auction has ended), the **extended-endtime** is increased by the **auto-extend-duration** and the original **endtime** remains unaltered as a record of the user set endtime as at lot creation.**.**

Example:

**endtime = 10:00**

**extended-endtime = 10:00**

**trigger-duration = 2 minutes**

**auto-extend-duration = 5 minutes**

**Bid** at 9:45 (no change to **extended-endtime)**

**Bid** at 9:55 (no change to **extended-endtime)**

**Bid** at 9:59 (**extended-endtime)**

### 

## Non-functional requirements

The following bullet point list is of the non-functional requirements and, for clarity, also re-iterates some of the above bidding logic:

* A bid, once it was placed, cannot be cancelled.
* A creator of a lot cannot bid on their own lot.
* A bid, once successful, cannot be retracted.
* When a bidder bids for an amount higher than 1 bidding-increment + current leading-bid, their bid-amount automatically becomes the pending-autobid.
* A bidder cannot bid again against themselves unless they have an autobid active and this second bid is higher than the autobid. E.g.
  + If their second bid is lower than their previous (current highest-bid) one, it is refused.
  + If their second bid is higher than their previous one but they have an auto bid active and their second bid is lower than the autobid, they’re told that they’re already in the lead.
  + If their second bid is higher than their active autobid, the autobid is increased.
* When a bidder bids for an amount equal to the existing pending-autobid, the previous bidder remains the leading-bidder (though their autobid is removed).
* When a bidder bids for an amount less than the existing pending-autobid, the highest-bid is set to current highest-bid + new bid + 1 bidding-increment.
* When a bidder bids for an amount more than the pending autobid, the highest-bid changes to current pending-autobid + one bidding-increment and their bid becomes the pending-auto bid, if it is more than the new highest-bid
* If it’s not the first bid, a bidder must enter an amount equal or higher to the bidding-increment + current leading-bid
* If it is the first bid
  + If there is a starting-price, a bidder must enter an amount equal or higher to the starting-price
  + If there is no starting-price, a bidder must enter an amount equal or higher to the bidding-increment
* if the endtime of a lot is reached and the leading-bidder is still below the reserve-price, they don’t win the lot
* Winning bidder earns item just by the next bidding-increment after the second highest-bid regardless of their pending-autobid amount

## Decision Table (with time extension logic not included)

The decision table is a well established technique (Gane & Sarson, 1979, 82-83)[40](#u1i76xrffvp) for recording complex business logic.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Rule:** | **R0** | **R1** | **R2** | **R3** | **R4** | **R5** | **R6** | **R7** | **R8** | **R9** | **R10** | **R11** | **R12** | **R13** | **R14** | **R15** | **R16** |
| C1 | First bid (Y-yes, N - No) | Y | Y | Y | N | N | N | N | N | N | N | N | N | N | N | N | N | N |
| C2 | this bid OP {L, LE, GT, GT1} highest bid | - | - | - | LE | GT1 | GT | L | EQ | GT1 | GT | LE | GT | GT | GT | GT | GT | GT |
| C3 | Leading bidder who? ( O-other, S-self) | - | - | - | O | O | O | S/O | S | S | S | S/O | O | O | O | O | S | S |
|  | Pending Auto-bid (Y-yes, N - No) | - | - | - | N | N | N | N/Y | N/Y | N | N | Y | Y | Y | Y | Y | Y | Y |
| C4 | Pending Auto-bid value (bid LE, GT, EQ) | - | - | - | - | - | - | - | - | - | - | - | L | EQ | GT1 | GT | LE | GT |
| C5 | Starting price value (LE, GT submitted bid) | LE | EQ | GT | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| ACTIONS | Digits show order of execution | L=lower, L1=lower with exactly 1 bidding increment, LE=lower or equal to, EQ=equal to, GT=greater, GT1=greater with exactly 1 bidding increment, OP=operator | | | | | | | | | | | | | | | | |
| AR | Reject Bid | X |  |  | X |  |  | X | X |  |  | X |  |  |  |  | X |  |
| A1 | Make this 'highest bid' |  | 1 |  |  | 1 |  |  |  | 1 |  |  |  | 1 | 1 | 1 |  |  |
| A2 | Make this 'pending autobid' |  |  | 1 |  |  | 1 |  |  |  | 1 |  |  |  |  | 2 |  | 1 |
| A3 | Make bidder 'leading bidder' |  | 2 | 2 |  | 2 | 2 |  |  |  | 2 |  |  |  | 2 | 3 |  |  |
| A4 | highest bid' increments by x or = |  |  | x=St Price |  |  | x=1 incr |  |  |  | x=1 incr |  | = (bid+incr) |  | x=autobid + 1 incr | x=autobid + 1 incr |  |  |
| A5 | Remove pending auto-bid |  |  |  |  |  |  |  |  |  |  |  |  | 2 | 3 |  |  |  |
| Excp ID | Exception Text |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | Bid amount less or equal to startingPrice= x | 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 | bid amount less or equal to current highest bid + biddingIncrement: x <= y + z |  |  |  | 1 |  |  | 1 | 1 |  |  | 1 |  |  |  |  |  |  |
| 2 | bid amount less or equal to biddingIncrement: x<y |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 | bidder already in control of lot |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 |  |
|  | Message Text Display |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | Your bid was accepted but equals an existing autobid |  |  |  |  |  |  |  |  |  |  |  |  | 3 |  |  |  |  |
|  | Your increased bid has been accepted |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 2 |

### Decision table example

A demonstration of how to read this table is given for rule 15 (R15) in the first row of the table:

Conditions

If this bid is not the first bid

And this bid is greater than the highest-bid

And the leading-bidder is not the bidder of this bid

And there IS currently a pending-autobid on this lot

And the amount of this bid is greater than the pending-autobid with more than one bidding-increment

Actions

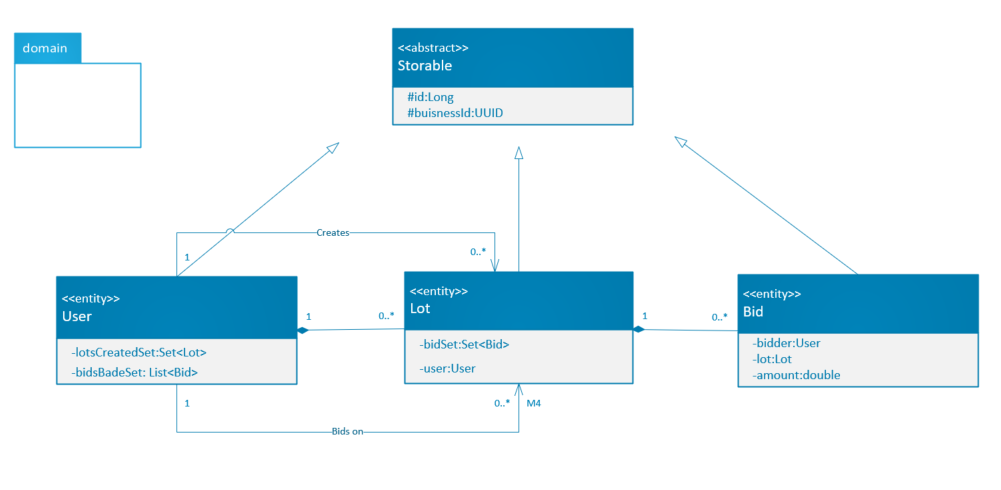
Make this bid the highest-bid

Make this bid the pending-autobid

Make this bidder the leading-bidder

Set the highest-bid to the pending-autobid + one bidding-increment

## **Diagram of Domain classes**

****

# Development Method and Implementation

This section details the process of implementing the Domain, Dao and Controller classes and the justifications for the various methods employed to achieve the most notable elements within them. It also highlights the techniques used to implement the underlying database tables that the above classes rely on as well as the front-end component that they drive.

## Domain classes

#### **Abstract class Storable:**

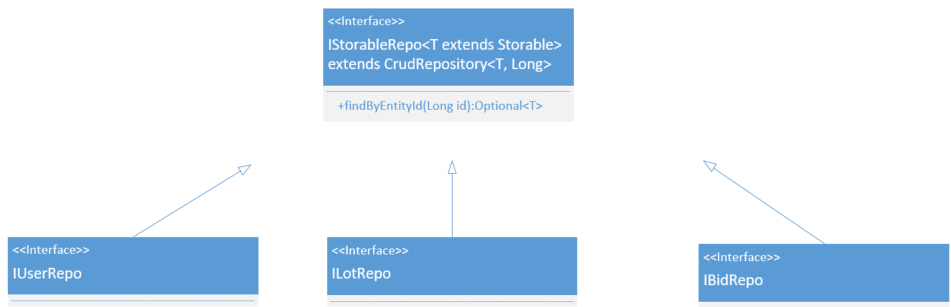
Storable has two attributes that are passed on to all its immediate subclasses. Storable denotes the fact that all of its subclasses can be stored in the database. The database in turn, has a table ‘storable’ with an additional ‘discriminator column’, *entity\_type* that shows to which of its concrete implementations, each record belongs to. This design is following the strategy described [here](https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/joined-table-inheritance.html)[41](#udd15vc70ogg)

|  |  |  |  |
| --- | --- | --- | --- |
| **Table: storable** | | | |
| **Column** | **Datatype** | **Attribute** | **Default** |
| id | bigint | PK | NOT NULL |
| business\_id | binary(255) | Unique | NULL |
| entity\_type | varchar(31) |  | NOT NULL |

The class has the Spring @Compnent annotation which lets Spring find this class when doing a component scan. It also has the @Entity jpa annotation which signifies that there should be a table maintained in the database, for this class. Next there is the [@Inheritance](https://www.baeldung.com/hibernate-inheritance)[42](#ytbzef9kb9cs) annotation declared, with a *joined* inheritance strategy type. Before we move on to explain why this inheritance strategy was favored out of the four possible ones, it would be useful to first present the Repository hierarchy, in this application.

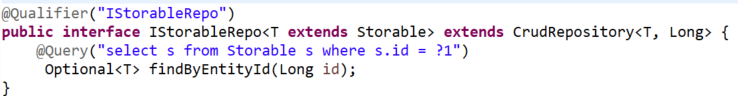
### Repository interfaces

The individual repositories in the repository hierarchy (of the type as referred to under number 3 of this [source](https://www.baeldung.com/spring-data-repositories)[43](#fuixdgp9yndi) and in section 1.1 of this [source](https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html)[44](#2guya5fermlp)) represent each, a table corresponding to an entity class in our application. The repositories are managed by Spring’s application context during the lifetime of the application. All that is required of the coder is to declare an interface type, have it extend Spring-data’s CrudRepository and Spring will create an implementation of it. A corresponding database table will be created for each repository interface declared and the persistence operations pertaining to the entity class a given repository represents, would then be invokable on that reposiotry.



In our case, as outlined in this [source](https://blog.netgloo.com/2014/12/18/handling-entities-inheritance-with-spring-data-jpa/)[45](#bfyk35hutl5z), we declare an intermediate IStorableRepo that represents Storable so that all the basic CRUD operations that are common to User, Lot and Bid can be invoked on an interface type of IStorableRepo. (The actual implementations of the CRUD operations are visible in Spring-data’s CrudRepository declaration.). The reason the intermediate repository cannot rather be of an interface type, e.g. IStorable, is that an interface alone does not guarantee that every implementation of it will be a managed @Entity type.

Accordingly in the code, the super repository and an example of one of the sub repository interfaces will be declared like this:





This would then be used like this:





Or:



To find a record, with the given id, of any entity that implements Storable.

In fact, even this would work:



This works because each record’s primary key in each subclass’ table is also a foreign key in relation to *storable’s* primary key column, both containing the same id value.

##### Inheritance strategy

So, returning to the reasons why *joined* strategy type was favoured, were the following:

* MappedSuperClass was ruled out as it would mean Storable cannot have its own table and in the hierarchy of repositories too, the operations that are common to all the subclasses of Storable, could not be generalized to be invokable on a super interface repository. And, as we shall later see, having this last capability reduces the amount of code in the *controller* and *service* components.
* Single Table, apart from the obvious discomfort that the bigger the hierarchy the bigger the table in which they’re all stored, with this strategy it also becomes clear that attributes cannot anymore have a *not-null* constraint for the table column corresponding to that attribute and a column must be present for all records in the table, even if that class in the hierarchy doesn’t even have that attribute. Given, the not-null constraint is only here to help one not to violate the business logic underpinning the data dependencies in the application, it seems careless to refuse using it.
* Table-Per-Class, was even less desirable due to each entity duplicating, in its table, the attributes inherited by its superclass. Also, now that each table declares its own ids, linking a subclass to their sibling’s tables by use of their id as a foreign key, does not make sense anymore because each table has its own ‘independent’ id column serving only a primary key purpose locally, and not also a forign key. If we wanted to also link this table to other sibling tables, more than one id column would be needed. But fundamentally, this design is to be avoided because as a result of this design, the underlying sql query when querying a child via the parent, contains a join on the result of a subselect, achieved by a UNION of all children/subclasses of this parent. This, depending on the size of the children tables can lead to performance issues. This [source](https://webdev.jhuep.com/~jcs/ejava-javaee/coursedocs/605-784-site/docs/content/html/jpa-inheritance-concreteclass.html)[46](#lo4ya9aoueji) explains why.
* Joined Table was also the one strategy that was familiar to the developer of this project from previous experience during other university modules.

This pattern isn’t exactly a Persistent Object Superclass as it is not Storable itself that provides the capability of storage but the Repository hierarchy that does that. Storable merely reflects persistence as a common capability of all its inheritors so that attributes and operations they all share because of this, can be recognized at compile time, on any *instance* of Storable, without explicitly knowing to which of the three concrete implementations the *instance* belongs.

##### Jackson annotations

For example, this design allows for controller endpoints, of types Post or Put, to simply specify an interface type in their json request body parameters and for Jackson, the tool that converts them to java objects, to still be able to deserialize them into their concrete types, so long as the *‘type’* attribute in the request body object, is specified, like this: {“type”:”lot”}. And, since all concrete classes that share some commonality should implement an interface or some common ancestor anyway, in our case it may as well be the abstract Storable class; especially given that the two attributes of *id* and *businessId* are indeed shared among all subclasses.

To demonstrate how we enable Jackson to distinguish which implementation of Storable a given incoming object is, we annotate the Storable abstract class with the following three annotations:



@JsonTypeInfo tells Jackson:

* Firstly, that the way the sender will identify a given json object to which concrete class it corresponds to, will be by using the logical ‘name’ of that corresponding concrete class inside the json object.
* Secondly, that the format of this information inside the json object will be formatted thus:type*:<value>* (as are the other attributes of the object - <*attr>:<value>*);
* Thirdly, that by default an incoming object is of type User (and doesn’t need to be identified - this is sensible since User is the top most parent from the POV of persistence.).

@JsobSubTypes.Type, provides the actual values for the logical ‘name’ mentioned above and for the mapping between it to the actual type (Class).

In summary, the nested annotation above dictates that json objects that can be instances of more than one entity types, should be identified with the form of {type:<entity\_type>} and for this to be included as part of their json serialized object, next to the other attributes of the actual object. In order for Jaskon to recognize the specified entity type, they are declared inside the @JsonSubTypes.Type annotations, as above.

##### Lombok

Next, we have the @Setter, @Getter and @ToString annotations, features of lombok library, that bring in the set(), get() and toString() methods respectively, for all properties in the class. Because id gets set by the persistence provider and doesn’t need to be set anywhere programmatically, the Setter for id was disabled with *@Setter(AccessLevel.NONE)*.

##### Primary key strategy

The Strategy for id generation was chosen to be Auto, which leaves the decision to the persistence provider and in our case the ‘identity sequencing’ strategy is chosen and maintained by a separate *hibernate\_sequence* table, containing a *next\_val* column, in the database.

The *id* of an entity is only set after an entity is first stored in the database, so it made sense to remove the *id* from the equals() and hashCode() methods since regardless of whether an entity is in a transient, saved or removed (from the database) state, if none of its other attributes have changed, it should still be equal in all three states. In other words, two objects with the same values for all attributes but the id attribute, should still be equal to each other.

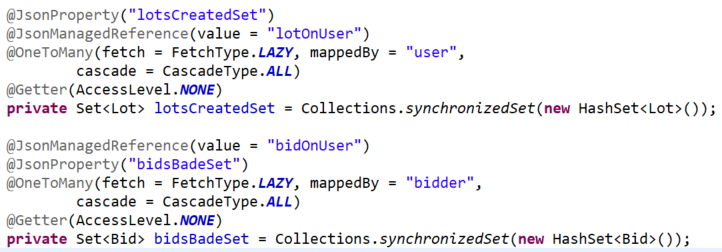
Moreover, since we use HashSet collections in this application, removing the *id* only from the hashCode method, will lead to a situation where hashCode says they’re equal and equals() says they aren’t. Although not a violation of the hashCode() contract, this is still undesirable as this would potentially cause a higher collision rate. Hence, the question remains, what makes them unique (The problem of equals() and hashCode() in entity classes is explored in detail in this [source](https://vladmihalcea.com/hibernate-facts-equals-and-hashcode/)[47](#atyzbcyod7r0) and also in this [source](https://thorben-janssen.com/ultimate-guide-to-implementing-equals-and-hashcode-with-hibernate/)[48](#xtir6q14te8g)).

To resolve the equality question, a unique *businessId* of type [UUID](https://www.baeldung.com/java-uuid)[49](#s6elo8eyincn), version 1, was introduced, with the annotation of @NaturalId above it, and now two objects with the same values for all their attributes are equal even if their *ids* differ. (The reason *id* wasn't simply replaced with businessId is 1. So that Hibernate can manage it, auto increment it etc. and 2. So that the *id* alone can tell us whether something has already been saved in the database once or not without always needing to actually load the record or query the repository. The *businessId* alone doesn’t provide that as this gets set right away, with object creation. Also, ideally, a businessId should be immutable, rendering updating the FK relation column of a child entity impossible.[50](#3yqckp2vt4ne)) However, as far as the database is concerned the criteria for disproving equality (e.g. if, after we store object A, we try and store a yet transient object B who has the values for all its attributes, apart from the id, the same as A) is more stringent, in that so long as two objects have the same *businessId*, even if the *ids* differ, they’re still considered equal.

#### **User extends Storable:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Table: user** | | | |
| **Column** | **Datatype** | **Attribute** | **Default** |
| id | bigint | PK, FK ref ‘storable’ (id) | NOT NULL |
| username | varchar(255) |  | NULL |
| password | varchar(255) |  | NULL |

The user entity has a small number of attributes and operations and also includes a builder to construct a user piece by piece. The two attributes worth exploring are the *lotsCreatedSet* of type Lot and *bidsBadeSet* of type Bid.



##### Thread safety

First it’s worth noticing that the reason a set was chosen is so that when adding a new lot or bid, we needn’t be concerned about duplication. Since, order is irrelevant in our design of the application and removal of a bid or a lot is also not specified in the requirements, by using a set, we save ourselves having to check everytime a new lot is added to the collection, if an equal value exists already, as the set doesn’t allow for duplicates. Now that the set is also synchronized, by using Java’s Collections.synchornizedSet static method to wrap the set and make it synchronized, we also remove the risk of a race condition that arises in a multithreaded scenario. E.g. the threat of a situation where two threads try to modify the set at exactly the same moment, trying to add different lots, and not all of the items (lots or bids) make it into the set.

Figure 1 below, is the code that tests thread safety, entirely based on [(Bugayenko, Y. 2018)](https://www.yegor256.com/2018/03/27/how-to-test-thread-safety.html)[51](#xdkk3nay2t7o):

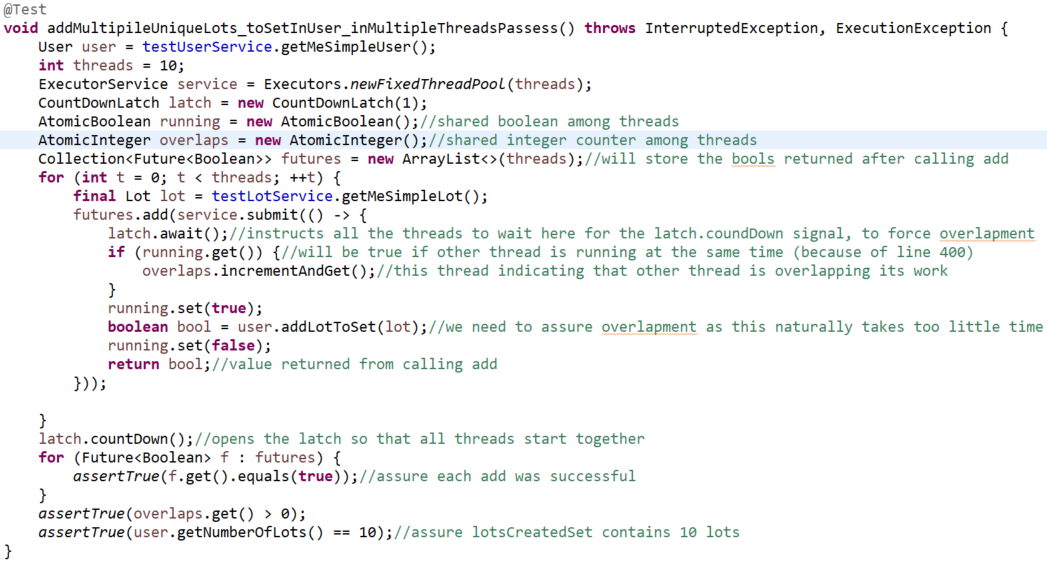


Figure 1

To explain the annotations above these sets:

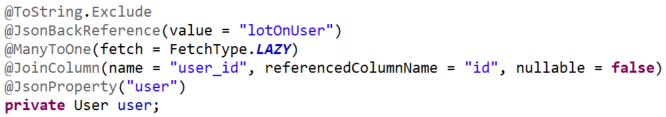
##### Json annotations

The @JsonProperty points Jackson’s object mapper from the attribute in a json object to its corresponding field, in the class. @JsonManagedReference together with its pair @JsonBackReference have a similar effect to lombok’s @ToString.Exclude for the toString() method in that it prevents Jackson entering into an infinite loop during serialization due to lotsCreatedSet containing lots and each of these lots containing a reference to their users and each user continuing a reference to lotsCreatedSet and so on.

The decision to make the child (i.e. the collection) managed as opposed to the parent was so that all entities could recursively, be displayed on the user object, in a Get request for example, instead of one having to remember the user *id* and go query for the children objects to see all those children that are related to this user.

##### ManyToOne associations

The @ManyToOne annotations, as explained at length in this [source](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping/)[52](#cjb3ngr2qo7m), and its inner properties tell Hibernate that one user can be associated with many lots and that the way a given lot is related to their user in the database, is by the ‘user’ reference inside the Lot class - this is achieved by the *mappedBy* property. In the database, this corresponds to the lot table, having a column for the user (*user\_id*) in which a user is actually referenced by the primary key in the record of that user, inside the user table. The ‘user’ attribute in the Lot class is decorated like so:



##### Fetch strategy

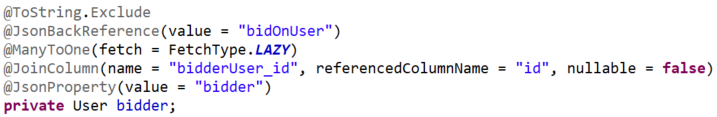
The fetch type dictates whether Hibernate should load relations eagurly or lazily and use proxies meanwhile. All the sources advocate for laziness and although for the size of this application, loading eagurly wouldn’t have made much of a difference on performance, still there’s no obvious benefit for doing it as the number of relations are small. This is especially the case here where we’re having a repository for each entity to load that entity directly as well as a super interface to load all entities from. We therefore need to worry less about an operation being invoked on a not yet loaded object via its parent as we’re more likely to load that child directly. That’s why we too followed the advice to have the children lazily loaded, with their parents.

##### Cascade strategy

With regards to the cascade type declared on the parent in relation to its child, the preferred choice is that all operations cascade so that e.g. if a user is saved, it’s related lots are saved with it; but the cascade type declared on the child is left to default which is, to do nothing. The downside is that everytime a child is encountered and we don’t know whether its parent is already stored (e.g. through a Get request) a check should be done first to see if its related parent (and so on recursively up the hierarchy) is not yet in the database; as otherwise if that parent cannot be null, an error would be thrown complaining that the object we’re attempting to save has a reference to a still transient object.

On the other hand if we’d tried to set on the child, cascadeType to *‘persist’* - to persist a parent automatically via the child, every time we’d try and save a child whose parent is already in the database, an error would be thrown complaining that we’re trying to save again an entity that is already in the database. Had we used Hibernate’s [sessions](https://www.baeldung.com/hibernate-session-object-states)[53](#wbah1z64dqx8), a saveOrUpdate method would’ve gotten round this but given the size of this application, using sessions seemed an overkill.

Similar to user in Lot.class, the *bidder* attribute in the Bid class is decorated like so:



Note that the *user* attribute in the User class and the *bidder* attribute in the Bid class, together break the conceptual many to many relationship that exists between Lot and User in that a lot can be associated with many users by way of *bidding* and one user can be associated with many lots by way of *creating.* So whilst finding all a given user’s lots would merely require to retrieve all lots from the lotsCreatedSet, finding all the bids for a given user, for a given lot, would require to filter that user’s bids by the lot desired, in the bidsBadeSet which is in the User class.

#### **Lot extends Storable:**

The following table illustrates the columns, data types and constraints for the lot table in the database:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table: lot** | | | |
| **Column** | **Datatype** | **Attribute** | **Default** |
| id | bigint | PK, FK ref ‘storable’ (id) | NOT NULL |
| auto\_extend\_duration | bigint |  | NULL |
| bidding\_increment | double |  | NOT NULL |
| description | varchar(255) |  | NULL |
| user\_id | bigint | FK ref ‘user’ (id) | NOT NULL |
| leading\_bidder\_user\_id | bigint | FK ref ‘user’ (id) | NULL |
| auto\_bid\_id | bigint | FK ref ‘bid’ (id) | NULL |
| start\_time | datetime(6) |  | NOT NULL |
| end\_time | datetime(6) |  | NOT NULL |
| extended\_endtime | datetime(6) |  | NULL |
| highest\_bid | double |  | NULL |
| starting\_price | double |  | NULL |
| reserve\_price | double |  | NULL |
| trigger\_duration | bigint |  | NULL |
| title | varchar(255) |  | NULL |

##### 

##### 

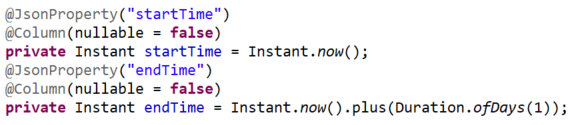
##### Spring dependency injection

The User owns a number of sensible attributes recorded on it some of which take on default values, either from the application.properties file, using the Spring’s @Value annotation, or from within the source code itself. For example, whilst title and and startingPrice have simple string values in the application.properties file, the values for triggerDuration and autoExtendDuration, coming in from the application.properties file, get parsed and instantiated to instances of Java’s [Duration](https://www.baeldung.com/java-period-duration)[54](#siioe160jv8) class, with the help of using [SpEL](https://docs.spring.io/spring-boot/docs/2.1.12.RELEASE/reference/html/boot-features-external-config.html#boot-features-external-config-conversion-duration)[55](#x3llsjgc2z6y) inside the @Value, like this:



It’s also possible to inject dependencies via the ‘post constructor’ method using the @PostConstruct annotation, as we do for the *clock* attribute. But, because this (as with field injection) is only a solution for when the object is instantiated with the default constructor (as would be the case e.g. when using the @Autowired for Spring and @Entity for Hibernate), care has been taken for fields that cannot remain null, to be set either inside the non-default constructors (as with *clock*) or explicitly in field declaration (as with *start* and *end times*, and the *sets*).

Another reason why for *startTime* and *endTime*, field initialization was chosen always to be the case, was due to ease of manipulating the type before setting the attribute as opposed to parsing an expression that does that, from the application.properties file. The downside is that if we wish to change e.g. the default duration of the endTime, this now has to be done from within the source code, something which is best avoided. Note, the Lot Builder doesn’t expose an extendedEndTime() method; instead extendedEndTime gets set to endTime in the constructor/s of Lot. Following is a code snippet of the *startTime* and *endTime* declarations:



Static attributes don’t get initialized at all with field injection and instead need setter injection. (though setter injection has the advantage that they work also when objects are created manually - that is objects constructed by parameterized constructors). A demonstration of this can be seen with the *bidSoftExcepFactory* attribute of typeExceptionsCreateor in the Lot class. Lombok’s global class @Setter and @Getter also have no effect on them and instead the attribute needs direct @Setter/@Getter decoration declared directly above them.

Similar to lotsCreatedSet in the User class, bidSet in the Lot class records all bids placed on this lot. A bid record in the bid table in the database is mapped to its associated lot by the column *lot\_id* that corresponds to the *lot* field inside the Bid class. This is provided by the combination of annotations above the *bidSet* field in Lot and the *lot* field in Bid. For a bid to be added to the set, it has to first get past the placeBid() method that defines the logic for a successful bid.

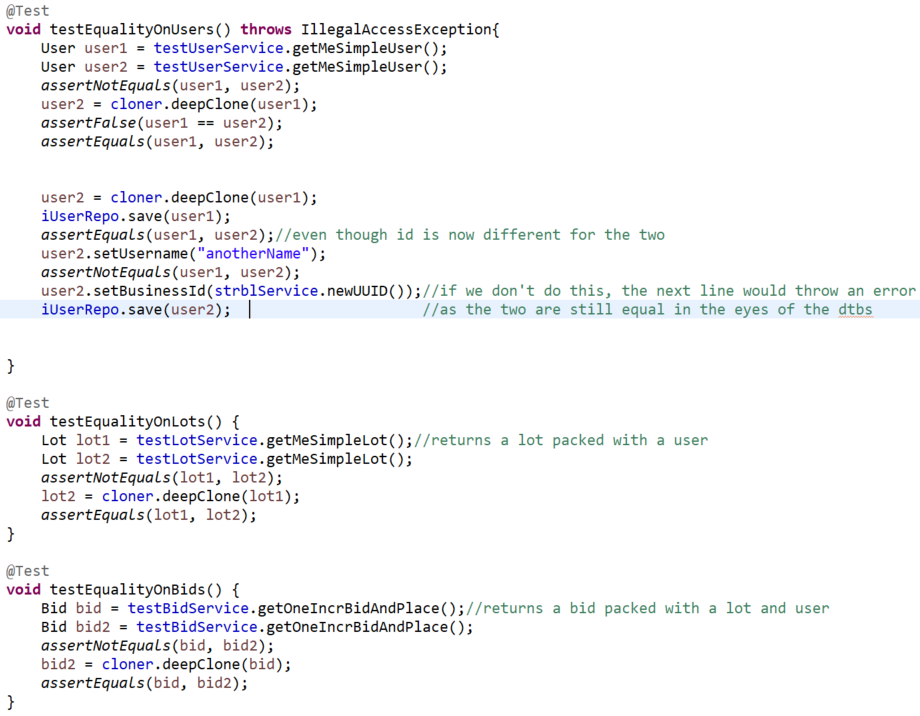
##### hashCode()

The problem of infinite looping crops up again in the *hashCode()* of the Lot class. Because a Set is used, if *user.hashCode()* is left within the *hashCode()* of the Lot class, the user’s *hashCode()* will in turn be invoked eagurly on each element of its *lotsCreatedSet* leading to an infinite loop; *user.hashCode()* was therefore removed from the *hashCode()* of the Lot class. A similar decision was taken with regards to *bidder.hashCode()* ( -> *bidsBadeSet* in User) and *lot.hashCode()* (-> *bidSet* in Lot) in Bid class.

##### equals()

Considering the *equals()* methods, although not an infinite loop problem at first glance, does require modification in a similar spirit to *hashCode()* in order for the *equals()* to return the correct answer, when the two objects are indeed equal. Because the *equals()* is not eagerly invoked on the collections within the parent class, two child objects that really **are** equal will show unequal because the *equals()* invoked on the parent attributes that reside within each child will return false. So long as the parents’ *equals()* do not eagerly check their collections, this is justified because two parent objects whose collections’ contents have not been checked against each other should not be equal. On the other hand if the *equals()* of the parents were to be modified so that it ***is*** eagerly invoked on each element of the collection, again an infinite loop would occur. The solution here therefore was the same: To remove *bidder.equals()* (-> bidsBadeSet in User) and *lot.equals()*  (-> bidsSet in Lot)from the *equals()* in Bid class. And, to remove the *user.equals()* (-> lotsCreatedSet in User) from the Lot class.

The following three test methods demonstrate what was just discussed as well as the id vs *businessId* from the database’ point of view.



##### OneToOne associations

The two attributes, leadingBidder and pendingAutoBid, although not conceptually one to one associations since one bidder can of course have many pendingAutoBids - and be the leadingBidder - on multiple lots, have been modelled this way because this application does not keep a reference to say a collection of pendingAutoBids on the bidder/user object. This is however easy to extract on the database level, should we wish to, by realizing that each lot stores a relation to its leadingBidder and pendingAutoBid, if there is one.

##### Operation specification in Lot

placeBid(Bid bid) algorithm logic description in pseudocode

public Optional<Lot> placeBid(bid){

assure bid is not null

round the amount of bid to nearest bidIncrement//in a real application we'd want to relate back to the bidder the modified amount

if (bid.bidder.equals(owner of lot)

throw exception(“can’t bid on own lot”);

else if (there is a leadingBidder on this lot AND the leadingBidder.equals(the bidder of this bid) AND the amount of this bid is higher than the highestBid but lower or equals to the pendingAutoBid (if there's one))

throw exception("you already control the lot");

try{

addBid(bid)

if there is no pendingAutoBid{

bump the highestBid up by one biddingIncrement;

if (the amount of bid is higher than the new highestBid)

set the pendingAutoBid to this bid;

set the leading bidder to the bidder of this bid;

}

else {

if the amount of bid is less than the pendingAutoBid

set the highestBid to the amount of this bid + one bidding increment

else if the amount of this bid is equal to the pendingAutoBid

{ set the highestBid to the amount of the pendingAutoBid;

remove the pendingAutoBid;

log a message explaining that although the bis was accepted, the amount of this bid equals an existing autobid;

}

else {

if( the leadingBidder.equals(the bidder of this bid)

log a message saying that their pendingAutoBid increase was accepted;

else

set the highestBid to the pendingAutoBid amount + one biddingIncreement;

set the leadingBidder to the bidder of this bid;

if (the amount of this bid is less or equal to the new highestBid)

remove the pendingAutoBid;

else

set the pendingAutoBid to this bid;

}

}

}

Catch (a lot has ended exception)

rethrow it;

Catch (a bid is too low exception)

rethrow it;

Catch (a general exception)

Return an empty optional of type lot;

Return an optional of this lot //(if this point is reached - all was successful)

}

private void addBid(bid){

now = the current wall clock time;

try{

checkBidHighEnough(bid.amount);

if now < extendedEndTime

Add the bid to bidSet;

else

Throw exception("lot has ended");

if isInTriggerPeriod(now)

set extendedEndTime to the extendedEndTime + autoExtendDuration;

}

catch (a bid is too low exception) {

rethrow it;

}

}

private void checkBidHighEnough(bid){

set isHigher = false;

If (highestBid > 0.0)

isHigher = the amount of bid is more or equals to the highestBid + one biddingIncrement;

else{// no bids placed yet

// there's startingPrice and bid is less OR there's isn't startingPrice and bid

// is lower than 1 increment

if (the startingPrice > 0.0 and the amount of bid is less than the startingPrice)

throw a bidTooLow exception explaining that the amount is lower than the startingPrice;

else if (the startingPrice == 0.0 AND the amount of bid is less than (one biddingIncrement)

throw a bidTooLow exception explaining that the amount is lower than the biddingIncrement;

else

Set isHigher to true;

}

If (!isHigher)

throw a bidTooLow exception explaining that the amount is lower than the highestBid + one biddingIncrement);

}

#### **Bid extends Storable:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Table: bid** | | | |
| **Column** | **Datatype** | **Attribute** | **Default** |
| bid\_id | bigint | PK | NOT NULL |
| bid\_amount | double | NN | NOT NULL |
| bidder\_user\_id | bigint | FK ref ‘user’ (id) | NOT NULL |
| lot\_id | bigint | FK ref ‘lot’ (id) | NOT NULL |

The Bid.class has three attributes, bidder of type User, lot of type Lot and amount of primitive type double. Again the usual annotations decorating the class. The @ToString includes an indicator to also call the superclass’ *toString()* method. The @PrimaryKeyJoinColumn, specifies the name of the column (id) in the bid table that links a given record in the bid table to its associated record in its superclass’ (Storable) table, by use of the primary key (also *id*) column in the *storable* table. @DiscriminatorValue, links all the ‘Bid’ values in the discriminator column (*entity\_type*) in the *storable* table, with the Bid class. And, an annotation not yet explained, the @JsonCreator above the parameterless constructor tells Jackson which constructor to use when instantiating this object on e.g. remote API requests.

## Controller, Service and DTO classes

**Controller, DaoServices and DTOs (Data Transfer Objects)**

The controller, and specifically the ServletInitializer class, with the help of the DaoServices is what allows queriers to create, update, read and delete entities to/from the database. The controller will map the request endpoint urls to the methods that will get executed when the given endpoint is hit in a web browser. In turn, depending on how complicated or specific the requested work is, the method will either pass on the work to the service instance responsible for carrying out the work pertaining to the object in question (i.e. User, Lot or Bid) or the controller will take care of it itself by using a general super interface repository of type Storable, without needing to know which specific implementation of Storable it is we're dealing with. The services in turn will each contain a repository instance of the entity type they’re supposed to be servicing and use this to resolve the request before returning results or acknowledgement to the controller. The services have a dependency on the DTO classes which are objects used for exposing and accepting a subset of the attributes of the true entity objects used in the business domain. By ‘exposing’ and ‘accepting’ we mean in terms of which of the attributes belonging to a domain entity, an external querier can request to update or read. In this application, we only used DTOs to restrict what an actor can ‘update’ and not what they can ‘read’. In plain terms, if a querier wishes to update a user, they send in a userDto object, that comes pre-set only with only those attributes we choose to allow updation, via Put and we use a mapper to extract the data from the dto and set the corresponding attributes in user to the data extracted from the userDto. DTOs in turn rely on mapperStructs, which are automatic, compile time created classes, that do the actual translation and modification between the dto and the user. The following diagram shows the relations between the controller, services, repositories and dtos.

#### Controller, Services and DTO Class diagram

When inspecting the ServletInitilizer class, we can see that the @RestController annotation decorates the class. This annotation is a specialized version of @Component and is also what tells Spring that each method contained within this class exposes an endpoint to handle a request and returns a serialized response body. The IDaoService interface’s implementers need to be annotated with @Qualifier because they’re all, at compile time, declared to be of type IDaoService<Storable>; and in order for Spring’s bean management to know at runtime to which one of the concrete implementation to instantiate them, we declare factory methods that produce the service implementers as managed Spring beans. We then, on top of the declaring variables in the ServletInitializer class (the objects that we wish should at runtime refer to whichever actual implementor), we specify a @Qualifier annotation containing the name of the method to be called at the time, the surrounding class, ServletInitializer, is instantiated. These methods, specified in the @Qualifier annotations, are called automatically due to the @Autowired annotation adjunct to them.

### Endpoints for Remote Procedure Calls for CRUD Operations

For instructions of how the remote procedure calls are carried out in order to execute any of the CRUD operations on the business entities, please consult [appendix 2](#_wzjpwnqx8wd7).

#### MapStruct Object Mapper

The Object Mapper interface BiddingSoftMapper is declared in the domain package and contains methods annotated with @BeanMapping whose properties dictate how to deal with null properties in the dto, that is, properties that the querier hasn’t provided a value for or has set to null. Whether to set the properties of the entities these dtos represent properties, also to null or to ignore them. Note that the attributes of primitive type *double* are here, in the DTOs, declared as their wrapper type *Double* so that they can be checked for nulls. Each method declared in the biddingSoftMapper interface deals with a separate DTO type’s translation from the DTO to the class’ that DTO represents. The method’s formal parameters receive the dto to get the property values from and the entity to set the property values to, that is, the source and the target. An actual implementation of this interface together with its declared methods is created inside the domain package within target/generated-sources/annotations/etc. folder, at compile time (or mvn package stage). This is enabled by a plugin in the pom.xml file, inside the annotationProccessorsPaths:

<path>

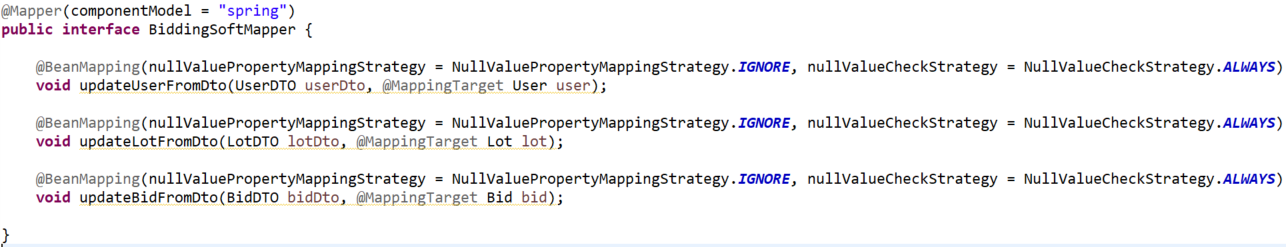
<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>${org.mapstruct.version}</version>

</path>

The following is a screenshot for the declaration of the object mapper interface:



#### Encapsulation

Here would be an appropriate place to talk about the issue of public setters and getters with regards to the entity’s properties. Ideally, even if mutability is rampant, that is fields are not declared final, setters and getters should contain some logic that limit modifications to the fields, else they shouldn’t be public as in this application. And so apart from the few methods that were crucial for proving by the test methods that the business logic works as intended (e.g. the getUser() and getBidder() methods), this approach would’ve been followed. But, because the mapper requires a public means to access the counterparts to the DTO’s attributes in the entity classes, all setters for the fields the DTOs includes, have therefore been declared public. Since the focus here was more on proving the possibility of the basic CRUD operations via endpoints -- and in particular the possibility of partial updates enabled by the use of the DTO pattern -- than on immutability and visibility within the project/module itself, the less desirable approach with regards to visibility, was chosen.

#### Operation Specification in Services

LotDaoServiceImpl persistEntity(Lot lot) algorithm logic:

public String persistEntity(Lot lot){

StringBuilder = new StringBuilder();

if (lot is not yet associated with lot.getUser())

add lot to the user's lotsCreatedSet;

if(lot.getUser() is not yet in the database){

save the lot's user to the database;

append to the stringBuilder the database id of the user;

}

else{

save just the lot to the database;

}

append to the stringBuilder the id of the newly created lot;

return stringBuilder.toString();

}

BidDaoServiceImpl persistEntity(Bid bid) algorithm logic:

public String persistEntity(Bid bid){

StringBuilder = new StringBuilder();

if(bid's lot is not yet in the database)

if (bids' lot is not yet associated with its user)

add bid's lot to the user's lotsCreatedSet;//doesn't add lot to the database; just associates the two

if(bid's lot and lot's user are both not in the database){

save the lot and user to the database;

append to the stringBuilder the database id of the user and the lot;

}

else if (only the lot is not in the database){

save the lot to the database;

append to the stringBuilder the database id of the lot;

}

if(bid's bidder is not yet in the database){

save the lot's bidder (as a user) to the database;

append to the stringBuilder the database id of the bidder;

}

add this bid to the bidsBadeSet of the bidder;

else{

save just the lot to the database;

}

append to the stringBuilder the id of the newly created lot;

save this bid to the database;

append to the stringBuilder the id of the newly created bid;

return stringBuilder.toString();

}

# Error handling

## Custom exceptions

Error handling, by use of *try* and *catch* blocks or custom exceptions are appearnet in all the places in the application where operations that may cause or throw errors are present; in the controller, the services and in the Lot class. bidSoftExcepFactory of type ExceptionsCreator in the Lot class is a managed bean factory that is used to create with it, instances of BiddingSoftExceptions’ subclasses, everytime we need one during the specification of the operations within the above mentioned components or classes. The subclasses of BiddingSoftExceptions represent the business errors which are thrown upon illegal operations one might attempt when interacting with the application and in particular with instances of Lot class.

BiddingSoftExceptions extends RuntimeExecpeptions, a decision taken due to the fact that these exceptions, on the side of whoever uses this code, are not compile time avoidable but also, once discovered, not hard to fix. That is, the only solution to these exceptions is that whatever the user has done incorrectly, in the way they invoked a given operation or the parameters they passed to it, should not be done and be corrected on the next attempt. E.g. if a user tries to bid after the lot has ended, there’s nothing they can do at compile time for this exception not to be thrown. To say they can is nonsensical. The way to fix that is not to do it. This [source](https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/)[56](#f39y0i5ta616) helped in reaching this decision. Each custom exception holds a String and id for its type of exception and to get the message related to the exception displayed, one can do *exception.getCause().getMessage()*.

# Logging

[Logback](https://www.baeldung.com/logback) is used with settings in two .xml files, one for src/main and another for src/test and the log files are generated/exported to the *logs* folder in the root folder of the app:

* In src/main, root logger is logging at debug level into files and custom loggers log to stdout and file, up to *info* level only.
* In src/test logback.test.xml, root logs to files, at debug level, and custom loggers log to both stdout and file - again up to *info* level only.
* Both use rollovers - that is new log files at every execution. The file names contain information identifying to which execution it refers to. src/main to files names of the form *logs-date;time.log* and src/tests to file names of the form *testlog-date;time.txt*.
* Generally, messages printed via custom loggers and explicitly from the source code are colored by the use of ansi color codes (log.info(ANSI\_RED + “text” + ANSI\_REST[57](#3btpudti55v6)) -> red) to distinguish them from logs generated from root or automatically. Colore support was enabled by inserting [*http://www.mihai-nita.net/eclipse*](http://www.mihai-nita.net/eclipse) in *help* -> *install new software* in eclipse IDE. See [link](https://github.com/mihnita/ansi-econsole)[58](#xcncd8zhb753).
* Note, currently there’s no measure to delete log files. They have to be deleted manually.

# Testing

The test methods are broadly divided in four categories:

* The consistency in the state of the domain entities at the application logic level
  + E.g. equals() and hashCode() methods
  + E.g. checking thread safety when adding lots to a user
* Persistence and integrity constraints at the database level
  + E.g. seeing if storing a user stores its lot with it
  + E.g. seeing if deleting a lot, leaves its user in the database
* Autowiring dependencies and Spring behaviour
  + E.g. checking that an autowired object has its autowired dependencies injected/present
* The logic of the bidding engine
  + E.g. checking that the application does what it should (creating lot, placing a bid, setting autoBid and other functional requirements)
  + E.g. assuring whether it doesn’t do what it shouldn’t (exceptions, rejecting bids for whatever reason etc.)
* That the exposed endpoints of the application are working
  + E.g. checking that an entity can be created or updated

Spring behaviour that is required globally as well as tests for the remote endpoints are included in the BiddingSoftwareApplicationIntegTests class. All the other tests are found in the UnitTests class. The two classes can be run separately (run as JUnit test in Eclipse) but running BiddingSoftwareApplicationIntegTests class also runs automatically UnitTests class. Note, running BiddingSoftwareApplicationIntegTests class, like running the application (as a Spring boot app) for the first time, causes the database tables to be created, if they don’t exist already, though the schema with the name specified in *application.properties* has to exist.

Running UnitTests, requires the tables to be there already. That’s why if the application is not started at least once in the normal way or through running BiddingSoftwareApplicationIntegTests class, and UnitTest class alone is executed, an InvalidDataAccessResourceUsageException, complaining that the tables/data queried don’t yet exist, will be thrown.

Something else to look out for: running BiddingSoftwareApplicationIntegTests starts up the application on port 8080 (if declared so as it is now in application.properties file) but if the application is already running, (started in the normal way - as a Spring-boot app from main()) an error will be thrown, complaining that port 8080 is already in use. The solution would be to first kill the application and then re-run BiddingSoftwareApplicationIntegTests class.

Use was also made of the @Before and @Order annotations to guarantee the order of some test methods execution. E.g. testing the ‘delete all entities’ endpoint should be done after there are records already present there and should the database be queried again after the ‘delete all’ method was executed, it should first be repopulated with data.

## Testing Tools

The following frameworks, libraries and techniques were made use of in the test files: Junit.jupiter, SpringBootTest annotations, Hamcrest asserts and apache HttpClient for testing the http methods. FieldUtils from *commons.lang.reflect* has also come in very useful, to demonstrate how mutation for the purpose of testing would be achieved if a given attribute **were to be** final or its setter **were to be** declared ‘private’.

## All test methods

For a detailed list of all the test methods created for this project, and their descriptions, see [Appendix 3](#_ycb9j23cf6ez).

### Testing classes diagram

### Test Results

## Real time testing

#### Overloading now()

To address the issue of testing a system designed to run in real time either the test data would need to be generated relative to the current time of day or a way to override the system clock, within the lot class, would be required and the test data would be fixed. The decision was made to go down the latter route because generating the data relative to the current time of day would be more complex. A Bealdung [article](https://www.baeldung.com/java-override-system-time)[59](#39j1kjn6avjq), titled Overriding System Time for Testing in Java, describing a variety of methods to override the system time suggests that the preferred approach (section 4.1) is to overload the *Instant.now(Clock)* method with a fixed clock instance, which is the approach we’ve taken. That is, to reflectively swap, during testing, the lot instance’s clock attribute with a fixed clock so that when *placeBid()* calles *now()*, based on that clock, it gets the intended overridden time.

#### Changing *System.currentTimeMillis()* globally

Approach number 3. in the above article, which changes what *System.currentTimeMillis()* returns when it is called, was rejected. This would have avoided tempering with any of the lot’s instance’s attributes. The downside of this approach is that a) it involves aspect oriented programming (and implementing this in a non-aspect project - calls for aspectj annotations) and b) in order to decide from which methods, and when, it should return something else, as opposed to something that changes how *System.currentTimeMillis()* behaves globally, would have involved much more work.

#### Passing a clock into placeBid()

Another approach that was initially tried but quickly rejected was to have placeBid() take a clock object, whenever a bid is placed, even in production. This was decided against because a) again it would mean designing the production code for the sake of testing and b) the realization, that making that method private so that it can not be used ordinarily and then testing the public method that uses it, would have led one back to stage one, with the question of how to get the modified clock in without a regular user/actor being able to do so, unresolved. This, because of a wide consensus that private methods shouldn’t be tested directly. So, the chosen solution was therefore to overload now() with a fixed clock, as explained above.

As an aside, it’s worth noting that Instant, e.g. what clock.now() returns, is ignorant of Daylight saving time; changing it to a [ZonedDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html)[60](#id77dt6i4ola) would have needlessly added more work to make it compatible to work with the other instance attributes of Lot of type Duration. Instant and Duration are more general and carry just enough information as they need, for the purpose of this application, and are workable with each other too well, for them to be specialized anymore than they are. If it had been the case of displaying the timings to an actual user, converting it to ZonedDateTime or even [LocaDateTime](https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html)[61](#dzv7hqo1h1vs) would have obviously been important.

# Conclusions

## Achievements

With the exception of ‘variable bid increment’ and ‘database integrity in multi user bidding’, listed in the bullet-point list of functionalities in the proposal document (BSc Final Year Project Form (2019/2020)) made for this project (*Objectives* section), the aims and objectives set out in that list of the proposal, have been achieved and other functionality that was not stated in the proposal was also added. The project proposal states:

*Core objectives will include:*

* *Model, code and test, in object oriented development style, the complex logic, appropriate checks and error handling involved in functionalities such as:* 
  + - *Placing a bid*
    - *Auto bidding*
    - *Variable bid increments*
    - *End-time extension*
    - *Wall clock testing*
    - *Database integrity in multi user bidding*
* *To connect a hierarchy of java classes together with a persistence database framework, for the purpose of storing a user’s actions and related data.*
* *To have the software also work via web browser, online.*

A total of 56 tests, consisting of Unit and Integration tests are passing successfully and more features[62](#byacvxxuzvix), especially on the front-end aspect of the application, exist for which tests were not written but are present and have been manually tested.

The plan for a GUI component if time permits was dropped, and instead a web front-end was chosen, out of a conscious decision to develop this application in a microservice style which by default is not meant to be used though a desktop GUI if it’s meant to be manually interacted with at all. Currently our understanding is that ideally, the data produced and consumed between microservices would be processed in an automatic and streamlined fashion by message buffers and exchanges, specialized tools to automate these processes and tie many small microservices together to build a coherent and bigger multi purpose application.

## Limitations

Variable bid increment was not incorporated as it was decided that every bid is automatically an auto bid. Including the possibility of placing an arbitrary amount bid would have complicated the project and the bidding logic in particular, a lot more. It was therefore not included. The constraint of database integrity in multi user bidding is irrelevant so long as the persistence layer bipasses the bidding logic and the application logic doesn’t feed into the persistence layer of the application. That is, the infrastructure for explicitly storing each change made to a user, lot or bid, throughout the bidding process, is there; it just hasn’t actually been implemented due to time constraints. Another, important aspect without which, this constraint would be incomplete is ‘thread safety’. In particular, placeBid(), an operation that intensively changes object state would need to be guarded for a multi user/lot scenario, possibly interacting in a multi-threaded environment, to assure that changes made to any of the entities are consistent in the persistence layer as well as in the application layer.

With regards to testing, only JUnit was used as the driving library supporting the tests conducted for this application. In this regard, the project failed to deliver upon the undertakings promised in the proposal. In addition, there is still plenty of code (or possible decision routes within the code) not covered by the existing tests and vital to be tested, should this application go into production for real use.

The application was also not developed and launched on or with Birkbeck’s titan server in mind, as originally set out in the proposal, out of a concern that the permission requirements and lack of ownership of the underlying technologies would present an obstacle to the configurational needs, the flexibility between the different layers and the tools used to support the coding, of this application.

## Future perspectives

A number of aspects of the project would benefit from further development if time had been available.

REST API: To truly implement a REST api with aggregate roots and for them to be hypermedia driven (as this [source](https://spring.io/guides/tutorials/rest/)[63](#idorth4rekte) demonstrates) the controller and services would need to be refactored and the logic of the CRUD operations altered to not bypass the application logic of e.g. placingBid(). This would give users a smoother and intuitive interaction with the application.

Database custom queries: At the movement, only the basic CURD operations are invokable on the repositories representing the entities in this application. To satisfy any meaningful and practical requirements in the application domain, such as ‘Retrieve all the bids for a specific lot' or 'all the lots related to a particular user’, custom queries should be drawn up and configured in a way that is easy for an external actor to invoke.These custom queries related to the business logic, would store the changes to the states of the objects as the application’s operations are running.

Drawing on from the previous point, assuring thread safety for the possibility of concurrent runs of the application being alive at the same time, would then logically follow for this application to stand a chance in the real world.

Immutability: Consideration was given to the importance of immutability in the current industry, for example, the popularity of the functional paradigm and of functional libraries introduced and used in the predominant OOP languages of today. However time did not permit focusing more on immutability as it would have meant having to medle oneself with the lower persistence detail and get more involved with Hibernate intricacies, capabilities and how it ‘does things’. This is unfortunate since the problem of violating encapsulation and in particular, having setters, with no limiting logic, public, would have been resolved automatically because if attributes cannot be mutated, setters are not needed.

Testing: As well as covering **all** existing code, more could have been done with other testing technologies like mockito or mockMvc to test objects and operations without actually having to create them all or specify exactly their inner logic. It would be good to learn more about the power these technologies have to offer and compare the advantages and differences between available testing frameworks.

Spring autowiring behaviour: Since we’re using Spring anyway, more could be done to actually make full use of the benefits Spring offers and in turn be more careful with default values for instance attributes (e.g. not to set it within source files) and by the same token be more creative with how to inject java custom types directly from the application.properties file.

# References

1. https://www.liveauctioneers.com/

2. https://www.the-saleroom.com/en-gb/how-to/how-to-bid-in-a-timed-auction

3. https://www.ebid.net/uk/help/selling-on-ebid/#

4. https://tarisio.com/buyers-sellers/buyers/bidding/

5. Bentaye. (2017, 11 8). *Simple online auction system in Java*. Stackexchange. Retrieved 5 4, 2021, from https://codereview.stackexchange.com/questions/179888/simple-online-auction-system-in-java

6. *krishnakarki195. (2016, 3 25). Bidding-System-In-Java. Github. Retrieved 5 4, 2021, from https://github.com/krishnakarki195/Bidding-System-In-Java/tree/master/src/main/java/com/bidding*

7.  *Bennett, S., McRobb, S., & Farmer, R. (2009). Object oriented systems analysis and design using UML (Third ed.). McGraw Hill.*

8. *Java Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/java/index.htm

9. *Microservice Architecture Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/microservice\_architecture/index.htm

10. *Eclipse Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/eclipse/index.htm

11. *Maven Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/maven/index.htm

12. *Gradle Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/gradle/index.htm

13. *Spring Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/spring/index.htm

14. *Spring Boot Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/spring\_boot/index.htm

15. *Groovy Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/groovy/index.htm

16. *UML - Object Diagrams*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/uml/uml\_object\_diagram.htm

17. *Design Patterns in Java Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/design\_pattern/index.htm

18. *Jackson Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/jackson/index.htm

19. *RESTful Web Services Tutorial*. (n.d.). Tutorialspoint. Retrieved 5 4, 2021, from https://www.tutorialspoint.com/restful/index.htm

20. (n.d.). Bealdung. Retrieved 5 4, 2021, from https://www.baeldung.com/

21. (n.d.). DZone. Retrieved 5 4, 2021, from https://dzone.com/

22. Jenkov, J. (n.d.). Jenkov.com. Retrieved 5 4, 2021, from tutorials.jenkov.com

23. (n.d.). docs.oracle.com. Retrieved 5 4, 2021, from https://docs.oracle.com/en/

24. *Guides*. (n.d.). spring.io. Retrieved 5 4, 2021, from https://spring.io/guides#tutorials

25. Rajput, D. (n.d.). Dineshonjava. Retrieved 5 4, 2021, from https://www.dineshonjava.com/

26. Gupta, L. (n.d.). Java Tutorials. Retrieved 5 4, 2021, from https://howtodoinjava.com/

27. Mihalcea, V. (n.d.). Vladmihalcea. Retrieved 5 4, 2021, from https://vladmihalcea.com/

28. Janssen, T. (n.d.). *Thorben Janssen*. Retrieved 5 4, 2021, from https://thorben-janssen.com/

29. (n.d.). Stackabuse. Retrieved 5 4, 2021, from https://stackabuse.com/

30.  Tarry, A. (n.d.). Andrew Tarry. Retrieved 5 11, 2021, from https://andrewtarry.com/

31. (n.d.). auth0. Retrieved 5 4, 2021, from https://auth0.com/

32.  (n.d.). Project Lombok. Retrieved 5 4, 2021, from https://projectlombok.org/

33.  http://logback.qos.ch/

34. (n.d.). Hamcrest. Retrieved 5 4, 2021, from http://hamcrest.org/

35. (n.d.). Stackoverflow. Retrieved 5 4, 2021, from https://stackoverflow.com/

36. (n.d.). Eclipse Foundation. Retrieved 5 4, 2021, from https://www.eclipse.org/

37.  https://www.postman.com/

38. Because at it is, these endpoints aren’t accessible over the internet

39. Stankowski, A. (2018, 11 25). *Immutable entities with Hibernate*. Allaroundjava. Retrieved 5 4, 2021, from https://allaroundjava.com/immutable-entities-hibernate/

40. Gane, C., & Sarson, T. (1979). *Structured Systems Analysis: tools and techniques*. Prentice-Hall.

41. *JPA - Joined Subclass Inheritance Strategy*. (2018, 1 7). Logicbig. Retrieved 5 4, 2021, from https://www.logicbig.com/tutorials/java-ee-tutorial/jpa/joined-table-inheritance.html

42. *Hibernate Inheritance Mapping*. (2020, 3 21). Baeldung. Retrieved 5 4, 2021, from https://www.baeldung.com/hibernate-inheritance

43. Aggarwal, S. (2019, 8 23). *CrudRepository, JpaRepository, and PagingAndSortingRepository in Spring Data*. Bealdung. Retrieved 5 4, 2021, from https://www.baeldung.com/spring-data-repositories

44. *Working with Spring Data Repositories*. (n.d.). docs.spring.io. Retrieved 5 4, 2021, from https://docs.spring.io/spring-data/jpa/docs/1.5.0.RELEASE/reference/html/repositories.html

45. *HANDLING ENTITIES INHERITANCE WITH SPRING DATA JPA*. (2014, 12 18). Netgloo. Retrieved 5 4, 2021, from https://blog.netgloo.com/2014/12/18/handling-entities-inheritance-with-spring-data-jpa/

46. *Chapter 2. Inheritance Strategy:Table per Concrete Class*. (2013). webdev.jhuep.com. Retrieved 4, 5, from https://webdev.jhuep.com/~jcs/ejava-javaee/coursedocs/605-784-site/docs/content/html/jpa-inheritance-concreteclass.html

47. Mihalcea, V. (2020, 8 14). *How to implement Equals and HashCode for JPA entities*. Vladmihalcea. Retrieved 5 4, 2021, from https://vladmihalcea.com/hibernate-facts-equals-and-hashcode/

48. Janssen, T. (n.d.). *Equals and HashCode with Hibernate - The Ultimate Guide*. thorben-janssen.com. Retrieved 5 4, 2021, from https://thorben-janssen.com/ultimate-guide-to-implementing-equals-and-hashcode-with-hibernate/

49. *Guide to UUID in Java*. (2021, 3 12). Bealdung. Retrieved 5 4, 2021, from https://www.baeldung.com/java-uuid

50.  Although despite keeping both, here *businessId* was still declared to be mutable, the reasons being: 1. for the purpose of equality testing and 2. even if *businessId* is not requested to be updated ‘directly’ (e.g. in a Put request), because it’s declared as a NaturalId, Hibernate seems to refresh it, everytime its object’s FK is updated and since our application generates these businessIds automatically upon creation of the object, without researching it nor spending enough time investigating it, our guess is that this is what causes the assignment of a new *businessId* to be attempted at each FK update. It was therefore left mutable.

51. Bugayenko, Y. (2018, 3 27). *How I Test My Java Classes for Thread-Safety*. Yegor256. Retrieved 5 4, 2021, from https://www.yegor256.com/2018/03/27/how-to-test-thread-safety.html

52. Gupta, L. (n.d.). *Hibernate one to many mapping annotation example*. HowToDoInJava. Retrieved 5 4, 2021, from https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping/

53. Dąbrowski, M. (2020, 10 18). *Object States in Hibernate’s Session*. Baeldung. Retrieved 5 4, 2021, from https://www.baeldung.com/hibernate-session-object-states

54. *Period and Duration in Java*. (n.d.). Bealdung. Retrieved 5 4, 2021, from https://www.baeldung.com/java-period-duration

55.  *Converting durations*. (n.d.). docs.spring.io. Retrieved 5 4, 2021, from https://docs.spring.io/spring-boot/docs/2.1.12.RELEASE/reference/html/boot-features-external-config.html#boot-features-external-config-conversion-duration

56. Gupta, L. (n.d.). *Java Checked vs Unchecked Exceptions*. Howtodoinjava. Retrieved 5 4, 2021, from https://howtodoinjava.com/java/exception-handling/checked-vs-unchecked-exceptions-in-java/

57.  (to reset the logger’s color for next time, to black)

58. https://github.com/mihnita/ansi-econsole

59. Baeldung. (2020, February 12). *Overriding System Time for Testing in Java*. Retrieved April 2, 2021, from https://www.baeldung.com/java-override-system-time

60. *Class ZonedDateTime*. (n.d.). docs.oracle.com. Retrieved 5 4, 2021, from https://docs.oracle.com/javase/8/docs/api/java/time/ZonedDateTime.html

61. *Class LocalDateTime*. (n.d.). docs.oracle.com. Retrieved 5 4, 2021, from https://docs.oracle.com/javase/8/docs/api/java/time/LocalDateTime.html

62.  E.g. ways of sending or updating objects.

63. *Building REST services with Spring*. (n.d.). Spring. Retrieved 5 11, 2021, from https://spring.io/guides/tutorials/rest/

64. https://www.oracle.com/uk/java/technologies/javase-downloads.html

65. https://docs.oracle.com/en/java/javase/15/install/installation-jdk-microsoft-windows-platforms.html#GUID-96EB3876-8C7A-4A25-9F3A-A2983FEC016A

66. https://www.git-tower.com/learn/git/commands/git-clone/

67. https://github.com/nkatz01/StuProj

68. https://git-scm.com/downloads

69. https://dev.mysql.com/downloads/windows/installer/8.0.html

70. https://www.sqlshack.com/how-to-install-mysql-database-server-8-0-19-on-windows-10/

71. https://github.com/nkatz01/StuProj/blob/master/java/biddingSoft/src/main/resources/application.properties

72. https://www.eclipse.org/downloads/

73. https://courses.cs.washington.edu/courses/cse373/18au/resources/eclipse-setup.html

74. https://www.codejava.net/ides/eclipse/change-java-compiler-version-for-eclipse-project

75. https://stackoverflow.com/questions/11130320/how-do-i-change-the-java-version-of-my-installed-project-facet-in-eclipse

76. https://maven.apache.org/download.cgi#:~:text=Apache%20Maven%203.6.3%20is,recommended%20version%20for%20all%20users.

77. http://mihai-nita.net/2013/06/03/eclipse-plugin-ansi-in-console/

78. https://howtodoinjava.com/automation/lombok-eclipse-installation-examples/

79. https://stackoverflow.com/questions/14059685/eclipse-maven-search-dependencies-doesnt-work#:~:text=Go%20to%20Window%20%2D%3E%20Prefrences%20%2D,will%20be%20ready%20to%20use.

# Appendices

## Appendix 1. Installation of the Development Environment

### Java

Download the latest (non-experimental) Java JRE (SE-11) from [here](https://www.oracle.com/uk/java/technologies/javase-downloads.html)[64](#bih3u1rexa0j). I downloaded the full JDK. Install it onto your machine and set the path variable as [appropriately](https://docs.oracle.com/en/java/javase/15/install/installation-jdk-microsoft-windows-platforms.html#GUID-96EB3876-8C7A-4A25-9F3A-A2983FEC016A)[65](#dmfdz6q3zxup).

### Git

If you’re intending to import this *project* into eclipse, on your local machine, via [cloning](https://www.git-tower.com/learn/git/commands/git-clone/)[66](#nw4rwh3qb0o6)a local git repository with the [remote](https://github.com/nkatz01/StuProj)[67](#xvluqzbe82j7), then obviously [downloading](https://git-scm.com/downloads)[68](#y50bj5oeuuvy)and installing git (and its Git Bash command line) is necessary.

### MySQL

1. [Download](https://dev.mysql.com/downloads/windows/installer/8.0.html)[69](#lii6qd52zniz)and [Install](https://www.sqlshack.com/how-to-install-mysql-database-server-8-0-19-on-windows-10/)[70](#y5vv59vbw4jw) mysql server with root as *user* and password: *bbkmysql* (settings are in the [application.properties](https://github.com/nkatz01/StuProj/blob/master/java/biddingSoft/src/main/resources/application.properties)[71](#xv7bg6q5sy3) file) of this *project*. The community-web version installer is enough. The instructions in the ‘Install’ link just mentioned will likely not look the same for you but the main components that are needed and useful are the Server, the Workbench, Connector/J, Documentations and Samples and Examples, and probably Notifier to allow the server to start automatically together with windows.
2. In the database, Create ‘biddingSoft’ schema

### Eclipse IDE

1. Download the latest version of [Eclipse](https://www.eclipse.org/downloads/)[72](#gj5taw2t1nr). The first couple of steps in this [link](https://courses.cs.washington.edu/courses/cse373/18au/resources/eclipse-setup.html)[73](#2lu1khual903) are all that’s required for eclipse installation, except that I chose ‘Eclipse IDE for Java EE Developers’ as I knew I was doing a web-application and also using JPA.
   1. You may also find it useful to follow the steps in the above link up to step 2: ‘Configuring checkstyle’ as they are mandatory anyway. Note, selecting the JRE/JDK may differ for you, depending on which version you downloaded in step 1. I recommend you download at least Java 11 to make sure it matches with the requirements, the dependencies of this project may have as well as the requirements of eclipse itself.
      1. Note: if you have already this *project* stored somewhere locally, you can go ahead now and choose your workspace to be StuProj\java. Else, you can always switch workspace later by doing File -> Switch workspace in Eclipse and then choosing the StuProj\java which you will have downloaded from the remote git repository.
   2. If you’re having trouble with eclipse switching between different versions of the JRE -- in case where you have multiple ones installed on your machine or perhaps because of what eclipse brings with itself -- I found it useful (after having imported or created my project) to navigate, in eclipse, [right-click] on project name -> build path -> configure build path -> Java Compiler -> [tick] Use compliance from execution environment -> [choose your preferred version]. As in this [link](https://www.codejava.net/ides/eclipse/change-java-compiler-version-for-eclipse-project)[74](#p3nclkp0h8a4).
   3. Another place that needed manual tempering is at [Project Facets](https://stackoverflow.com/questions/11130320/how-do-i-change-the-java-version-of-my-installed-project-facet-in-eclipse)[75](#bzr7d78er8wj) (which is found on the same level as Java Compiler). Select the appropriate version next to the Java facet.

#### Maven

You can of course download maven from [here](https://maven.apache.org/download.cgi#:~:text=Apache%20Maven%203.6.3%20is,recommended%20version%20for%20all%20users.)[76](#7wwuyjt528r4) as I did initially but it is now my understanding that eclipse includes already maven tooling and apart from a little configuration as we shall now see, nothing more is needed. For clarification, maven is a tool to automatically, build, package and manage Java projects. For instance, packaging the project as a jar or specifying a plugin that generates code, to be executed every time the project is rebuilt, becomes a very smooth process. Also, if you want to use a certain Java library, you can use maven to install that dependency/library from maven’s central repository into your local repository, which maven manages, and then maven will cleverly highlight that line of code which uses that library and help you choose the right *import* line for that piece of code. Similarly, all the different *starters* that are typically needed for a Spring (boot) project are managed by maven in what’s known as a pom file. A pom file works in a hierarchical structure, pulling in all the dependencies recursively, that your project depends on.

Note, if you’re using maven to manage your project, be careful not to easily accept suggestions prompted by eclipse itself, as they may confuse maven in the role that it is meant to play.

#### Eclipse configuration

In eclipse, from Help -> Eclipse Marketplace -> [search for] Spring Tools 4 (aka etc) 4.10.0 release. This will enable you to run the project as a Spring boot application by doing Run -> Run as -> Spring Boot App and then navigate to [*http://localhost:8080/*](http://localhost:8080/) on a web-browser to see the results. Or stopping it by hitting the red box underneath *Project*, top left in eclipse.

Note:

To stop the application from the command line, enter Ctrl+C and reply with Y to the prompt.

Or, do netstat -ano |findstr 8080 to get the process number for the task that is currently running on port 8080 and after that do *taskkill /F /pid processNumber* to kill that process and free the port.

#### Color

Whilst you’re at it, in order to see the colors of the text that the logger in this *project* is printing out to the console, it’s necessary that you download ANSI coloring plugin from Help -> Install New Software and click Add to add the following link http://www.mihai-nita.net/eclipse as on the top of the page in this [link](http://mihai-nita.net/2013/06/03/eclipse-plugin-ansi-in-console/)[77](#b9qi9h5vlcmz).

#### Lombok

Although the lombok dependency and configuration is already added to the pom.xml file, lombok requires a manual integration with eclipse. This is done by navigating to username\.m2\repository\org\projectlombok and executing the *java -jar lombok-1.16.18.jar* command in the CLI from this location. A pop up window would ask you to provide it with a link to your eclipse.exe executable which for me resides at this location myusername\eclipse\jee-2021-03\eclipse and after that, clicking instal/update would complete lombok integration with eclipse, allowing for the source files to recognize the @Setter, @Getter and @ToString annotations etc. The following [source](https://howtodoinjava.com/automation/lombok-eclipse-installation-examples/)[78](#80ef9f8i713a) is a step by step guide to installing lombok in eclipse.

### Import project

If this *project* is not imported yet, go ahead and from File -> Import -> Maven -> Existing Maven Projects and browse to StuProj\java (if you’re not there yet) to do -> Select Folder and the biddingSoft/pom.xml com..etc..SNAPSHOT:war file should be selected. (It’s also fine if biddingSoft is omitted and it starts from pom.xml etc. - that is you select initially the StuProj\java\biddingSoft folder.). You might see a lot of errors in the form of red crosses on the folder and package names but that will soon go away after we do a few tweaks and touches.

In order for maven to automatically help your searches and see dependencies in the central repository, it’s necessary to update the indexes. That’s done by going to Window -> Show View -> Maven -> Maven Repositories and when that ‘Maven Repositories’ tab opens, do Global Repositories -> Central (link etc.) -> [right click] -> [tick] Enable Full Index and do subsequently a Rebuild Index which is found on the same level. This [link](https://stackoverflow.com/questions/14059685/eclipse-maven-search-dependencies-doesnt-work#:~:text=Go%20to%20Window%20%2D%3E%20Prefrences%20%2D,will%20be%20ready%20to%20use.)[79](#g9a0g6214x6f) may help.

Another thing that’s important is to make sure that at Window -> Preferences -> Maven, "Download repository index updates on startup" is ticked, also mentioned in the previous link.

Finally, after having imported this *project*, right click on the project top folder -> Maven -> Update Project -> [select] Force Update of etc. and hit OK. This should remove all the rad flagged error crosses from the folder and package names.

## Appendix 2. Usage of CRUD web endpoints

To use any of the endpoints, it is necessary to first run the application. To check, via CLI, if the application is running do: *curl -v http://localhost:8080*.

### Usage

#### Get

To get a specific domain entity: (Note: Owner type entities are not shown on the Child side of the relationship.).

In postman do:

|  |  |
| --- | --- |
| <http://localhost:8080/getent>/<id> | of type GET, in the url bar |
| (where id is an existing id of a user, lot or bid) | |

#### Get All

For getting all existing entities from the database.

In postman do:

|  |  |
| --- | --- |
| [http://localhost:8080/](http://localhost:8080/getent)allents | of type GET, in the url bar |

#### Delete

For deleting a specific entity from the database.

In postman do:

|  |  |
| --- | --- |
| [http://localhost:8080/](http://localhost:8080/getent)delent/<id> | of type DEL, in the url bar |
| (where id is an existing id of a user, lot or bid) | |

#### Delete All

For deleting all entities in the database.

In postman do:

|  |  |
| --- | --- |
| [http://localhost:8080/](http://localhost:8080/getent)delents | of type DEL, in the url bar |

#### Post

To create an entity: (Note: not-null attributes cannot be omitted and the id of new objects cannot be decided by the querier. They are assigned by the application.)

|  |  |
| --- | --- |
| [http://localhost:8080](http://localhost:8080/getent)/create | of type POST, in the url bar |
| To create a new user:  {"type":"user"} -> (returns the <newUserId> for the newly created user)  To create a new lot based on the user just created:  {"type":"lot", "user":{"id": “<newUserId>” }} -> (returns the <newLotId> for the newly created lot)  To create a new bid based on the new lot just created:  {"type":"bid", "bidder":{"id":"<existingUserId>"} ,"lot":{"type": "lot","id":”<newLotId>"}} -> (returns the <newBidId> for the bid just created)  Note: bidder’s id isn’t referring to that of user | In the body  Note: The json objects can be sent In sequence with 3 separate requests as in the left or mixed in a variety of forms as per the various examples below. |
| Note: only lot and bid need their type specified with the type:object pair form (e.g. {“type”:”lot”} or {“type”:”bid”}); user doesn’t. This is enabled by the @JsonTypeInfo annotation in the Storable class - which recognizes ‘user’ as the default implementation of Storable. | |
| More possible ways of using this endpoint:  **default for new user**  {}  **New lot with existing user**  {"type":"lot", "user":{"id": “<existingUserId>” }}  **New lot and new user**  {"type":"lot", "user":{}}  **New bid with existing lot, user and bidder**  {"type":"bid", "bidder":{"id":"existingUserId"} ,"lot":{"type": "lot","id":"<existingLotId>"}}  **New bid with only existing user/s:**  {"type":"bid", "lot": {"type":"lot","user":{"id": "<existingUserId>"}}, "bidder":{"id":"<existingUserId>"} }  **With only existing lot creator:**  {"type":"bid", "lot": {"type":"lot","user":{"id": "<existingUserId>"}}, "bidder":{} }  **With only existing bidder:**  {"type":"bid", "lot": {"type":"lot","user":{}}, "bidder":{"type": "user","id": "<existingUserId>"} }  **With only existing lot and lot creator:**  {"type":"bid", "lot": {"type":"lot","id": "<existingLotId>"}, "bidder":{} }  **All entities new**  {"type":"bid", "lot": {"type":"lot", "user":{}}, "bidder":{} } | |
|
|
|

#### Put

Whilst Create, Read and Delete will, given the small size of the application, allow for a general and mixed approach between the three entities (e.g. creating a user alone or together with a lot at once), Update, due to the need of mapping the DTOs to the domain entities, will require separate Put requests for each. However, columns in the entities related to the entity currently updated, that indicate a relationship to this entity, such as FK, will get updated automatically, by way of updating the parent.

To update a specific ‘lot’:

In postman do:

|  |  |
| --- | --- |
| [http://localhost:8080/](http://localhost:8080/getent)update | of type PUT, in the url bar |
| To update the name of a user and add a relationship with an existing lot to them  {"type":"userdto", "id":"exsistingUserId" ,"username":"newName", "lotsCreatedSet": [{"type":"lot", "id":"exsitingLotId", "user": {"id":"exsistingUserId"}}] }  To update an existing lot’s relationship with their lot creator  {"type":"lotdto", "id":"existingLotId" , "user":{ "type":"user","id":"existingUserId"}}  To update an existing bid’s relationship with their bidder and lot  {"type":"biddto", "id":"existingBidId" , "bidder":{ "type":"user","id":"exisitingUserId"}, "lot":{"type":"lot", "id":"exsistingLotId"}} | |
| Note the use of userdto, lotdto and biddto. As in the other endpoints with entity types, by default, the type in this endpoint is *userdto*. Also, if an entity is updated via the entity’s name directly as opposed to its dto, as in the first example above, fields that are not provided in the object will be overridden with null by Hibernate (that’s one of the reasons for the use of DTOs in this application). | |

## Appendix 3. Tables of Unit and Integration test methods

### Unit testing methods

|  |  |  |
| --- | --- | --- |
| **Test method name** | **What it does** | **Rule** |
| newLot\_canBeSavedToDtbs() | Tests that a lot can be created and saved to dtbs | - |
| ableTo\_createBid\_AddLotToIt\_andSaveToDtbs() | Tests that a bid can be created, placed on lot and saved to dtbs | - |
| deleteBid\_leavesLotInDtbs() | Tests that deletion of bid doesn’t cascade to lot | - |
| deleteLot\_leavesUserInDtbs() | Tests that deletion of lot doesn’t cascade to user | - |
| deleteLot\_removesAllRelatedBids() | Tests that deletion of lot cascades to bids | - |
| addingSecondBidOnLot\_doesntAffectFirstBid\_indDtbs() | Tests that each bid is dependent of the other bids in the dtbs | - |
| whenLotIsCreated\_extendedEndTimeEqueals\_endTime() | Tests that at construction of a lot, extendedEndTime equals endTime | - |
| placeBid\_equalOrAfterEndTime\_ThrowsException() | Tests that bids cannot be placed at or after endTime of a lot | - |
| placeBidBefore\_triggerDuration\_EndTimeIsNotExtended() | Tests that placing a bid before the triggerDuration period, leaves the lot endTime as it is | - |
| placeBidWithin\_triggerDuration\_extendsEndTime() | Tests that placing a bid in the triggerDuration period, extends the lot endTime | - |
| placeOneBidIncr\_bumpsHighestBidUp\_byOneIncr() | Tests that one biddingIncrement bid bumps highestBid up by one biddingIncrement | R4 |
| bidWhenThereIsNoStartingPrice\_thatIsLowerThanOneBidIncr\_isRefused() | Tests that a bid which is below the biddingIncrement for a lot that doesn’t have a startingPrice, is refused. | R3 |
| bidWhenThereIsNoStartingPrice\_thatIsEqualToOneBidIncr\_isAccepted() | Tests that a bid which is equal to the biddingIncrement for a lot that doesn’t have a startingPrice, is accepted. | R4 |
| bidBelowStartingPrice\_isRefused() | Tests that a bid which is below the startingPrice for a lot that has a startingPrice, is refused. | R0 |
| bidEqualToStartingPrice\_isAccepted() | Tests that a bid which is equal to the startingPrice for a lot that has a startingPrice, is accepted. | R1 |
| bidAboveStartingPrice\_highestBidBecomesStartingPrice() | Tests that a bid which is more than startingPrice for a lot that has a startingPrice, causes the highestBid to be set to the startingPrice. | R2 |
| placeBid\_higherThanCurrentHighestBid\_changesLeadingBidder() | Tests that a bid that is higher than the current highestBid is accepted and changes the leadingBidder. | R4 |
| bidThatEqualsTo\_oneIncrMoreThanPrevBid\_doesNotbecomePendingAutoBid() | Tests that a bid that is equal to one biddingIncrement higher than the previous bid doesn’t become pendingAutoBid | R4 |
| bidThatIsOver\_oneIncrMoreThanPrevBid\_becomesPendingAutoBid() | Tests that a bid that is above to one biddingIncrement higher than the previous bid becomes a pendingAutoBid | R5 |
| bidThatIsBelow\_highestBidIsRefused() | Tests that a bid that is below highestBid is refused. | R3 |
| bidThatIsEqual\_highestBidIsRefused() | Tests that a bid that is equal to highestBid is refused. | R3 |
| bidAgainstSelf\_thatIsBelow\_highestBidIsRefused() | Tests that a leadingBidder who bids against themselves with a bid that is below highestBid is refused. | R6 |
| bidAgainstSelf\_thatIsEqual\_highestBidIsRefused() | Tests that a leadingBidder who bids against themselves with a bid that is equal to highestBid is refused. | R7 |
| bidAgainstSelf\_thatIsEqual\_oneIncrMoreThanPrevBid\_isAccepted() | Tests that a leadingBidder who bids against themselves with a bid that is equal to 1 biddingIncrement over highestBid becomes highestBid. | R8 |
| bidAgainstSelf\_thatIsAbove\_oneIncrMoreThanPrevBid\_becomesAutobid() | Tests that a leadingBidder who bids against themselves with a bid that is with 2 biddingIncrement over highestBid becomes pendingAutoBid. | R9 |
| bidThatIsBelow\_highestBid\_whenThereExistsAutobid\_IsRefused() | Tests that a bid that is below the highestBid when there is an autobid in place gets rejected. | R10 |
| bidThatIsEqual\_highestBid\_whenThereExistsAutobid\_IsRefused() | Tests that a bid that is equal to highest bid when there is an autobid in place gets rejected. | R10 |
|  |  |  |
| placeBid\_thatIsBelowAutoBid\_kicksHighestBidUp\_byOneIncr\_higherThanBid() | Tests that a bid that is below the pending autobid causes the highestBid to go up to one biddingIncrement higher than the bid. In effect, that the system is auto bidding on behalf of the owner of the pending autobid. | R11 |
| placeBid\_thatIsEqualToAutoBid\_autoBidIsRemoved() | Tests that a bid that is equal to the pending autobid causes the autobid to be removed | R12 |
| placeBid\_thatIsAboveAutoBid\_withOnlyOneIncr\_leadingBidderChanges\_andAutoBidIsRemoved() | Tests that a bid that is only one biddingIncrement higher than the pending autobid changes the leading bidder and removes the autobid | R13 |
| placeBid\_thatIsAboveAutoBid\_withMoreThanOneIncr\_becomesAutoBid() | Tests that a bid which is above the pending autobid with more than one biddingIncrement becomes itself the autobid | R14 |
| bidAgainstSelf\_thatIsOverHighestBid\_butBelowAutobidThrowsException() | Tests that a bid by a leadingBidding against themselves with the bid being lower than their autobid is refused. | R15 |
| bidAgainstSelf\_thatIsOverHighestBid\_andEqualAutobidThrowsException() | Tests that a bid by a leadingBidding against themselves with the bid being equal to their autobid is refused. | R15 |
| bidAgainstSelf\_thatIsOverHighestBid\_andOverAutobidThrowsException\_isAccepted() | Tests that a bid by a leadingBidding against themselves with the bid being higher than their autobid is refused. | R16 |
| addMultipileUniqueLots\_toSetInUser\_inMultipleThreadsPassess() | Tests that adding multiple lots from different overlapping threads to the lotsCreatedSet in the User class, succeeds and that no lot is lost | - |
| testEqualityOnUsers() | Tests the equals() method on users | - |
| testEqualityOnLots() | Tests the equals() method on lots | - |
| testEqualityOnBids() | Tests the equals() method on bids | - |
| testThatCallingEquals\_andHashCode\_doesntThrowStackOverFlow() | Tests that calling hashCode() and equals() on the three domain entities doesn’t result in a stackoverflow error | - |
| whenLotIsCreated\_autowiredAttributes\_areSet() | Test that setter injection works for properties of manually created lots (as opposed to via Spring) | - |

### Integration testing methods

|  |  |
| --- | --- |
| **Test method name** | **What it does** |
| removeAllTables\_andSetUpWiredLot() | An @Before utility method to truncate all the database tables from data populated in the previous run. This method also adds a lot and a bid to the autowired lot object |
| contextLoads() | Tests that Spring’s application context autowired the controller attribute of this class |
| test\_app\_is\_up() | Tests that the application was started and is running at the root endpoint |
| whenLotIsAutoWired\_autowiredAttributes\_areSet() | Tests that Spring’s autowiring of lots, using the default constructor, sets also those attributes of lot that are autowired using field and setter injection |
| testThatSuperRepo\_canReturnSubclass\_ofStorable | Tests that super repository, representing Storable, can be used of subclasses of Storable |
| testPostEntity\_forUser() | Tests the Post endpoint for a new user object |
| testPostEntity\_newLotWithExistingUser() | Tests the Post endpoint for a new lot object, where its linked user already exists |
| testPostEntity\_newBidWithExistingUser\_AndExistingBidder() | Tests the Post endpoint for a new bid object, where its linked lot and user already exists |
| assertWiredLot\_andItsBid\_areInDtbs\_andIdsAreNotNull() | Test that autowired lot and lot and bid that were added to it are in the repository/database |
| testGetOneEntity() | Tests the Get endpoint for a single entity from the dtbs |
| testUpdateUser() | Tests the Put endpoint for updation of a user in the database |
| testUpdateLot() | Tests the Put endpoint for updation of a lot in the database |
| testUpdateBid() | Tests the Put endpoint for updation of a bid in the database |
| testGetAllEnts() | Tests the Get endpoint for all entities from the dtbs |
| testDelOneEnt() | Tests the Del endpoint for deletion of a single entity from the dtbs |
| testDelAllEnts() | Tests the Del endpoint for deletion of all entities from the dtbs |

**~ END OF DOCUMENT ~**