Backend challenge

Job Fair 2023
by Aleksandra Petrović

Contact:
LinkedIn
GitHub
+381603613601
a.petroviic37@gmail.com

Auctions - project info & documentation

Intro

General
This is a standard Java Spring Boot project.

Data
For the database I used MySQL. The necessary steps to take in order to connect to the database are:

1. Import `mysql-connector-j` dependency in `pom.xml` file (already done ✅)
2. Run MySQL Server in the background
3. Edit the `application.properties` file to match your account
    a. Change `spring.datasource.username=` property
    b. Change `spring.datasource.password=` property

For easier database and transaction management I used Spring Data JPA which can also be found imported as a dependency in the `pom.xml` file.

Implementation

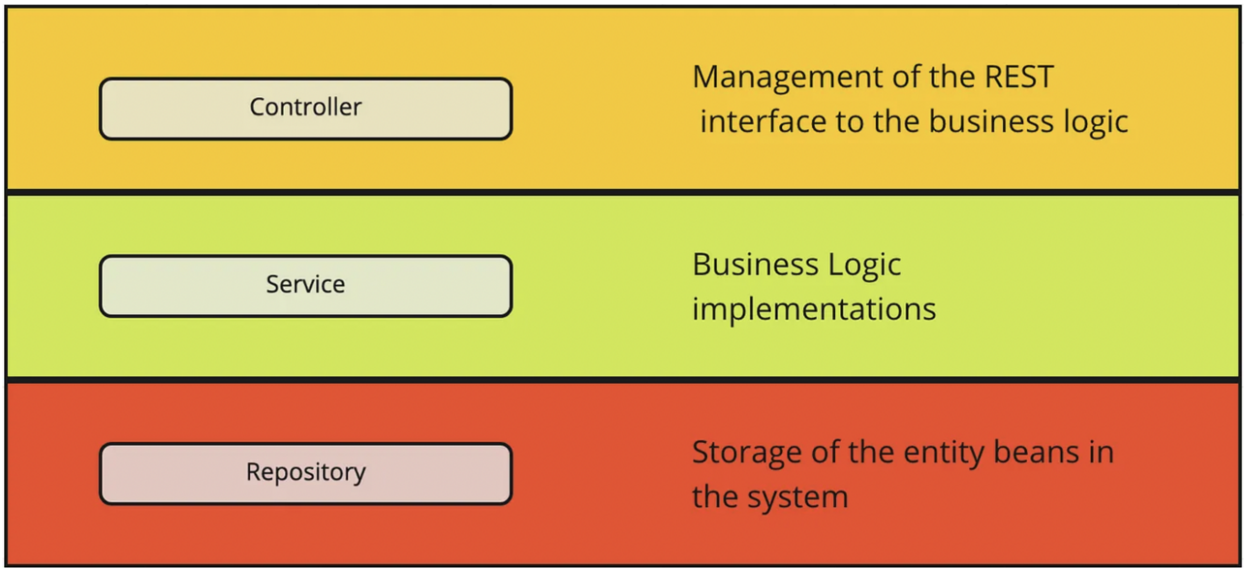
General

The goal of this project was to create an Auction Service that would handle all logic concerning auctions.

My focus was on creating a service that is:

- Reusable
- Scalable
- Highly cohesive
- Abstract

The app was built on a Controller-Service-Repository pattern in order to maintain the separation of concerns.


Controller
Management of the REST interface to the business logic
Service
Business Logic implementations
Repository
Storage of the entity beans in the system

Data

In a realistic scenario, this service would communicate with other services that handle the logic behind Users (Managers) and Players. Since I didn't have access to that data I needed to generate it myself.

I created my own pool of 20 users by simply running a "for" loop in `BootstrapData.java` class. I solved it this way because for this particular assignment there aren't any concerns about user history, level or any other characteristics.

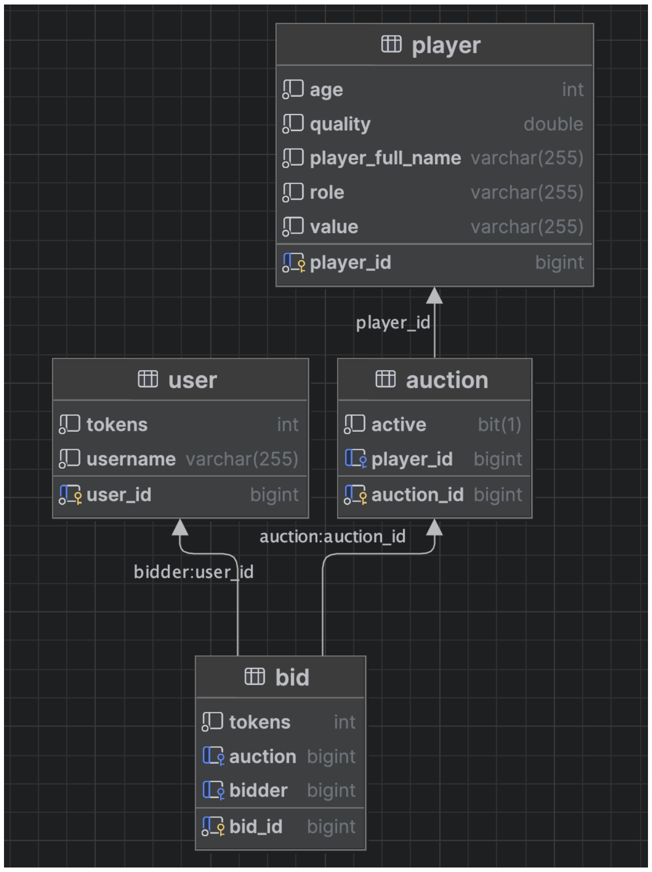
When it comes to generating players, I saw an opportunity to embed some realistic data to make it more interesting. So I searched kaggle.com for some cool football player datasets that I can use. I ended up finding this one, downloading it, cleaning it from unnecessary columns and preparing it to extract the data. I should also mention that I kept only the first 400 rows, since there was no need for more.

The columns that I copied to my entity were:

- `String playerFullName;`
- `String role;`
- `Integer age;`
- `Double quality;`
- `String value;`

I tried to make it look as similar to the player representation in Top Eleven as possible, so for example I scaled the quality of the player from a 1-100 scale to a 1-5 scale so it can fit the 5-star standard that is in the game.

For information about other entities and their relationships, here is the database schema:



Implemented Requirements

Scheduled events:

- Starting new auctions every minute
- Counting down time until the end of the auction

These events are implemented using `@Scheduled` Spring Boot Annotation.

Notifications:

- End of auction and the winner

- Placed bid on an auction
- Refresh new auctions generated

These notifications were implemented using `@Slf4j` Spring Boot Annotation that enables logging.

User actions:

- Getting all active auctions
- Joining an auction
- Bidding on an auction

All of the above mentioned functionalities have additional explanations in the actual code.

Restrictions

- Users cannot place a bid on an auction they didn't join first.
- Users cannot join an auction that is not active anymore.
- Users cannot bid on an auction that is not active anymore.
- Users cannot bid on an auction if they don't have enough tokens to match the current highest bid (+1).
- If users send a request to list all active auctions they automatically exit any auctions they previously joined.

If any of these events happen - the app will not crash, it will gracefully inform the user about the adequate restriction.

API

Because I wanted to simulate a process of authorization and security that I would normally implement in a realistic scenario - for every request a simplified JWT is necessary.
This means that in order to access these endpoints you need to create a request that contains Authorization part of the Header that looks like this:
"Bearer: [long: userId]"
For example: "Bearer: 4"

Endpoints

GET `/auctions/active`
- returns a list of all active auctions at the moment of requesting

GET `/auctions/{auctionId}`
- returns a single requested auction

GET `/auctions/{auctionId}/join`
- joins the given user in on a requested auction

GET `/auctions/{auctionId}/bid`
- places a bid on a requested auction in the name of the given user

Additional Information

It can be found inside the actual code. Some documentation and exact explanation of the logic and the specifics is right there.

Note: Please ignore the warnings considering DDL that come up at application startup. The reason is because in the `application.properties` file I used `spring.jpa.hibernate.ddl-auto=create-drop` because it gives me a clean slate every time I run the app and I could easily test functionalities. This could be easily fixed by using for example `spring.jpa.hibernate.ddl-auto=update`, but I left it this way because it makes more sense for an assignment like this.
It doesn't actually happen with the first run, but every other. Because it is trying to do a DDL operation on an entity that is dropped when the last run was terminated.

Thank you so much for your attention, Nordean! ⚽ 😄