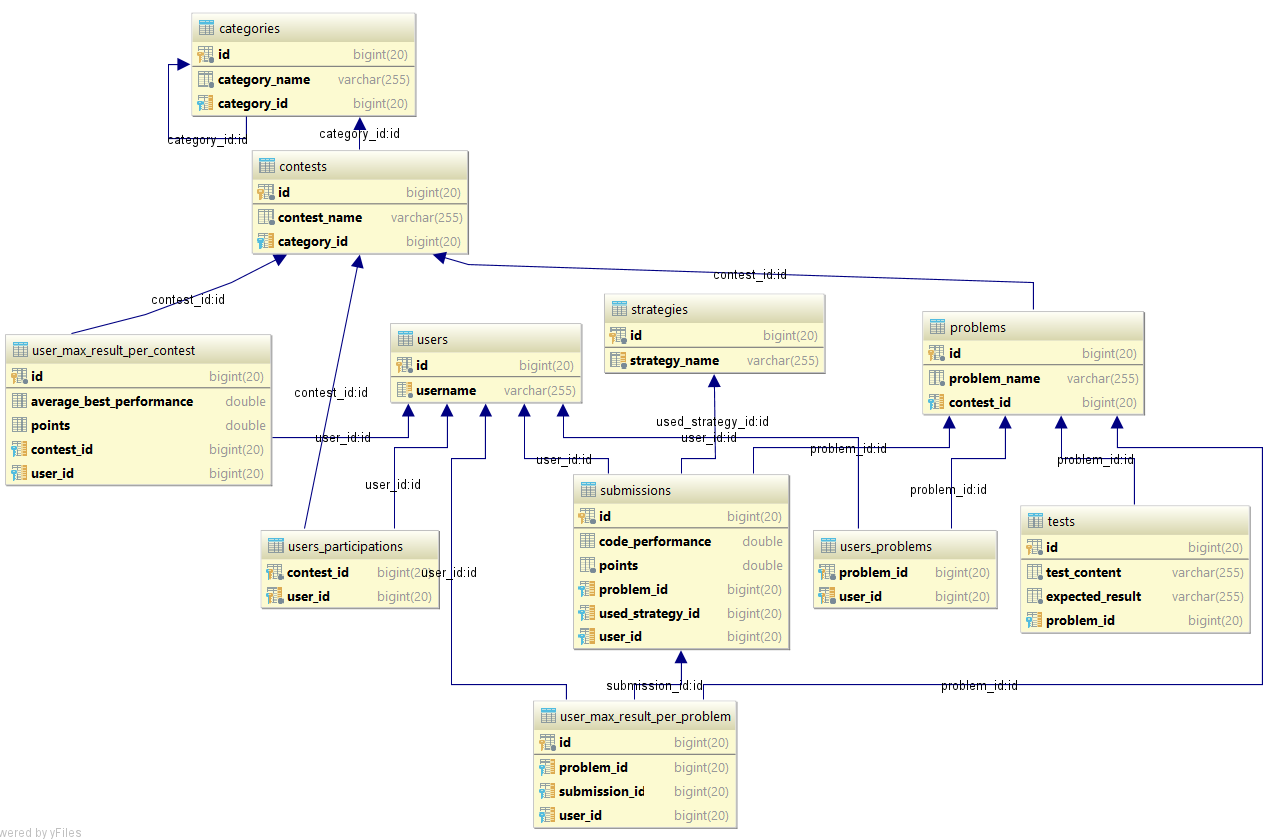
# Workshop: Creating large DB Part 2

This exercise is part of the [“Databases Frameworks” course @ SoftUni](https://softuni.bg/courses/databases-advanced-hibernate).

## Judge System

In the previous assignment we modelled the **structure** of our Judge System Database. Now it’s time to start writing some functionality logic and validation using that schema. For now, your database should have the following structure:



Download the skeleton from the course instance if you haven’t completed it in the previous lecture.

## Categories

Start by inserting the **categories first from the categories.json file**. Have in mind that a category cannot be inserted without it’s super category existing;

HINT: Create categories in the database recursively starting from the top category. If categories share the same super category, just add the new one to the subcategories of the existing one.

## Strategies

Proceed by inserting strategies. They are held in the **strategies.json** file.

## Contests Creation

All contests will be persisted from the **contests.json** file. Each contest belongs to certain category and allows many strategies to be used. Get each contest’s category from the database and assign it to it. Then, add the permitted strategies to each contest’s collection of strategy and persist the contest entity.

## Contest Enrollment

Import the **users.json** file. Write the functionality to allow users to **enroll** to contests. **Once a user has enrolled to a contest**, he can solve the problems in it, otherwise the application **should not allow** problems to be visible to him.

Contest enrollment is done by passing **contest** and **user** **id** to an enrolling logic. Consider that a user **cannot start practicing a contest that doesn’t exist or is not logged in**. If he is already enrolled to it, return an adequate message “**Used already has enrolled to contest**”.

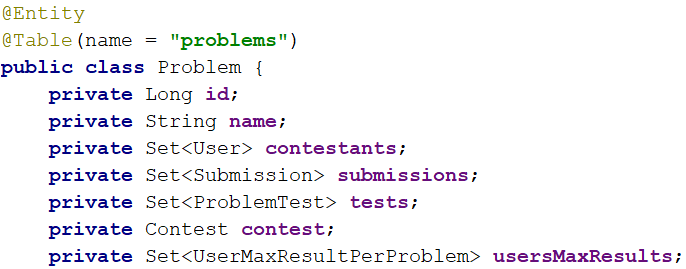
#### **HINT**:

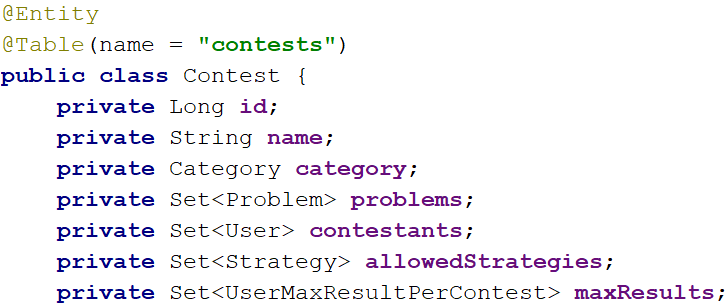
You can begin by creating an UserParticipation java object that will represent our user-contest pair from the database. Use the newly created object to map the xml file to the database. From this object, you can get information about the **user’s id**, **get him from the database** and **add the contest to his participations collection**, again by the contest id from the same object and persist the user.

## Problem Solving

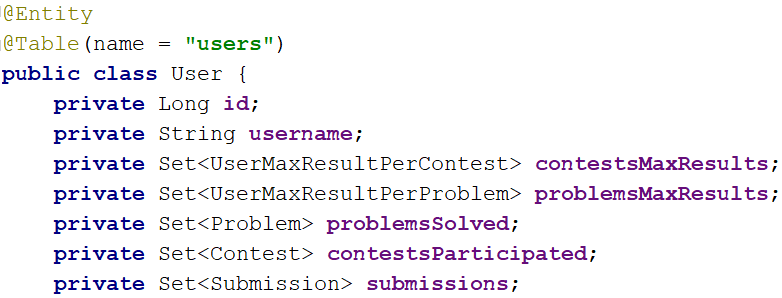
We’ve cleared that once a user has enrolled to a contest, he has access to the contest’s problems. Look at the way we’ve designed our **user**, **contest** and **problem** entity:

Each problem entity keeps information about the max results obtained for each user;





The same goes for our users, except that they’re optimized to give us info about the maximum results they have achieved for each contest and problem, so we don’t traverse entities unnecessary



With this having in mind, once a user submits a **submission, his maximum result for the problem and contest must be updated.**

### Submitting

Submitting is done **only if the user has access to the problem**, in other words - **is a contestant in the problem’s contest**. Each submission gives certain amount of points, which increase his maximum results accordingly if needed – **only if the current submission is a new maximum one** for the user to the current problem. If so, the UserMaximumResultPerContest for the corresponding problem is updated and with that the user’s maximum result for the contest.

#### HINT:

Each submission keeps information about the problem id and the user who made it. Get the user from the database and check if he is a contestant in the problem’s contest. If so, add the submission to the database and proceed to step 6, otherwise throw an adequate message like “**User is not contestant in contest!**”

## Result Update

Our entities keep information about the maximum results made with/towards them.

Once a user submits, his maximum results for the problem to which he submits a solution should be updated **only if** the new scored result is a maximum to this point.

HINTS: After each submission made for user, update the UserMaximumResultForProblem entity **if the new result from the submission is higher than the already existing one**. This means that the initial submit towards a problem is considered a maximum and the comparison starts from there. With each update on the UserMaximumResultForProblem, update the UserMaximumResultForContest if the new result is a top one for the contest. A result is a top one for the contest only if the points scored are higher than the maximum points scored to this point. **If there is a draw**, a maximum result is considered **the one with submission with best code performance**.

## Bonus Tasks

### Validation Rules

Add validation to each entity that we’ve created according to the following description:

#### Category

* **Name** – cannot be null; length must be more than 4 symbols; must start with capital letter

#### Contest

* **Name** - cannot be null; length must be more than 4 symbols; must start with capital letter
* **Category** – cannot be null; All contests are placed in certain categories

#### Problem

* **Name** - must be longer than 3 symbols; must start with capital letter
* **Contest** – cannot be null

#### Strategy

* **Name** – cannot be null; must start with capital letter; is **unique**

#### Submission

* **Code performance** – cannot be null; precision up to 3 digits after decimal point
* **Points** – cannot be null
* **Problem** – null is not allowed
* **Used strategy** – cannot be strategy that is not allowed by the contest; cannot be null
* **User** – null is not allowed

### Register

Write extra functionality so users can register in the system by giving the following information:

* **Password** – longer than **5 symbols**, containing at least **one special symbol** and **digit. Modify the user entity to add password**
* **Username** – longer than 4 symbols, **not starting or ending with special symbols. Write register logic according to the following criteria:**
* Users will have to input their password twice to confirm it. In case the passwords differ, they should not be able to login
* Apply validation to the entity’s fields. Give adequate exception messages if validation fails.
* Store the new user if all above steps are passed

### Login

After a user has **registered successfully**, he should be able to **login**, **if he already hasn’t**. Keep his credentials in an Application context, like previous exercises. He should be able to do the following actions in the assignment **only if he has logged in**.