Gamified Marketing Project

Specifications - User Features

An application deals with gamified consumer data collection. A user registers with a username, a password and an email. A registered user logs in and accesses a HOME PAGE where a “Questionnaire of the day” is published.

The HOME PAGE displays the name and the image of the “product of the day” and the product reviews by other users. The HOME PAGE comprises a link to access a QUESTIONNAIRE PAGE with a questionnaire divided in two sections: a section with a variable number of marketing questions about the product of the day (e.g., Q1: “Do you know the product?” Q2: Have you purchased the product before?” and Q3 “Would you recommend the product to a friend?”) and a section with fixed inputs for collecting statistical data about the user: age, sex, expertise level (low, medium high). The user fills in the marketing section, then accesses (with a next button) the statistical section where she can complete the questionnaire and submit it (with a submit button), cancel it (with a cancel button), or go back to the previous section and change the answers (with a previous button). All inputs of the marketing section are mandatory. All inputs of the statistical section are optional. After successfully submitting the questionnaire, the user is routed to a page with a thanks and greetings message.

Specifications - Scoring

The database contains a table of offensive words. If any response of the user contains a word listed in the table, the transaction is rolled back, no data are recorded in the database, and the user's account is blocked so that no questionnaires can be filled in by such account in the future.

When the user submits the questionnaire one or more trigger compute the gamification points to assign to the user for the specific questionnaire, according to the following rule:

1. One point is assigned for every answered question of section 1 (remember that the number of questions can vary in different questionnaires).

2. Two points are assigned for every answered optional question of section 2.

When the user cancels the questionnaire, no responses are stored in the database. However, the database retains the information that the user X has logged in at a given date and time.

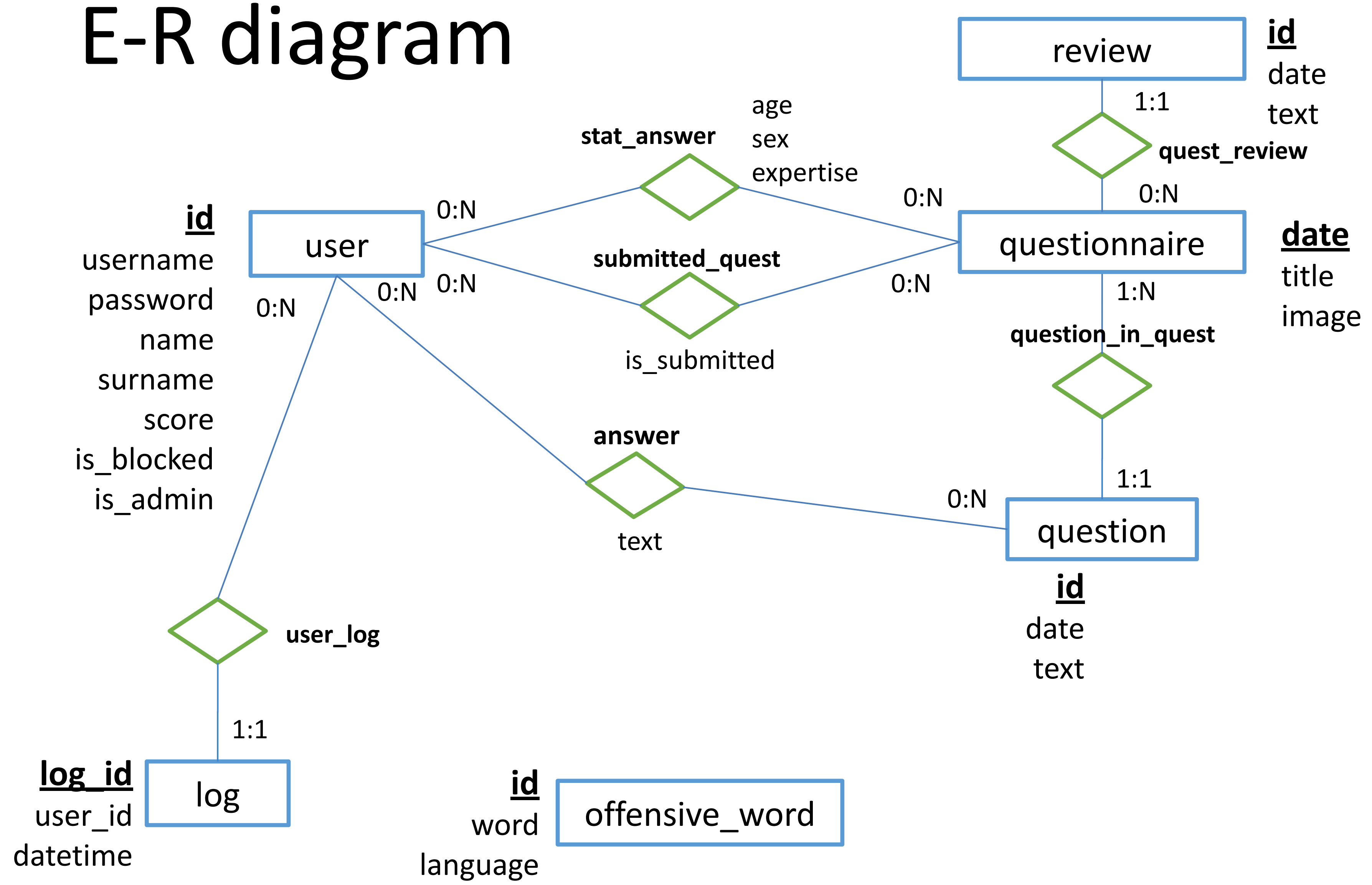
The user can access a LEADERBOARD page, which shows a list of the usernames and points of all the users who filled in the questionnaire of the day, ordered by the number of points (descending).

Specifications - Admin Features

The administrator can access a dedicated application on the same database, which features the following pages:

- A CREATION page for inserting the product of the day for the current date or for a posterior date and for creating a variable number of marketing questions about such product.
- An INSPECTION page for accessing the data of a past questionnaire. The visualized data for a given questionnaire include:
    - o List of users who submitted the questionnaire.
    - o List of users who cancelled the questionnaire.
    - o Questionnaire answers of each user.
- A DELETION page for ERASING the questionnaire data and the related responses and points of all users who filled in the questionnaire. Deletion should be possible only for a date preceding the current date.

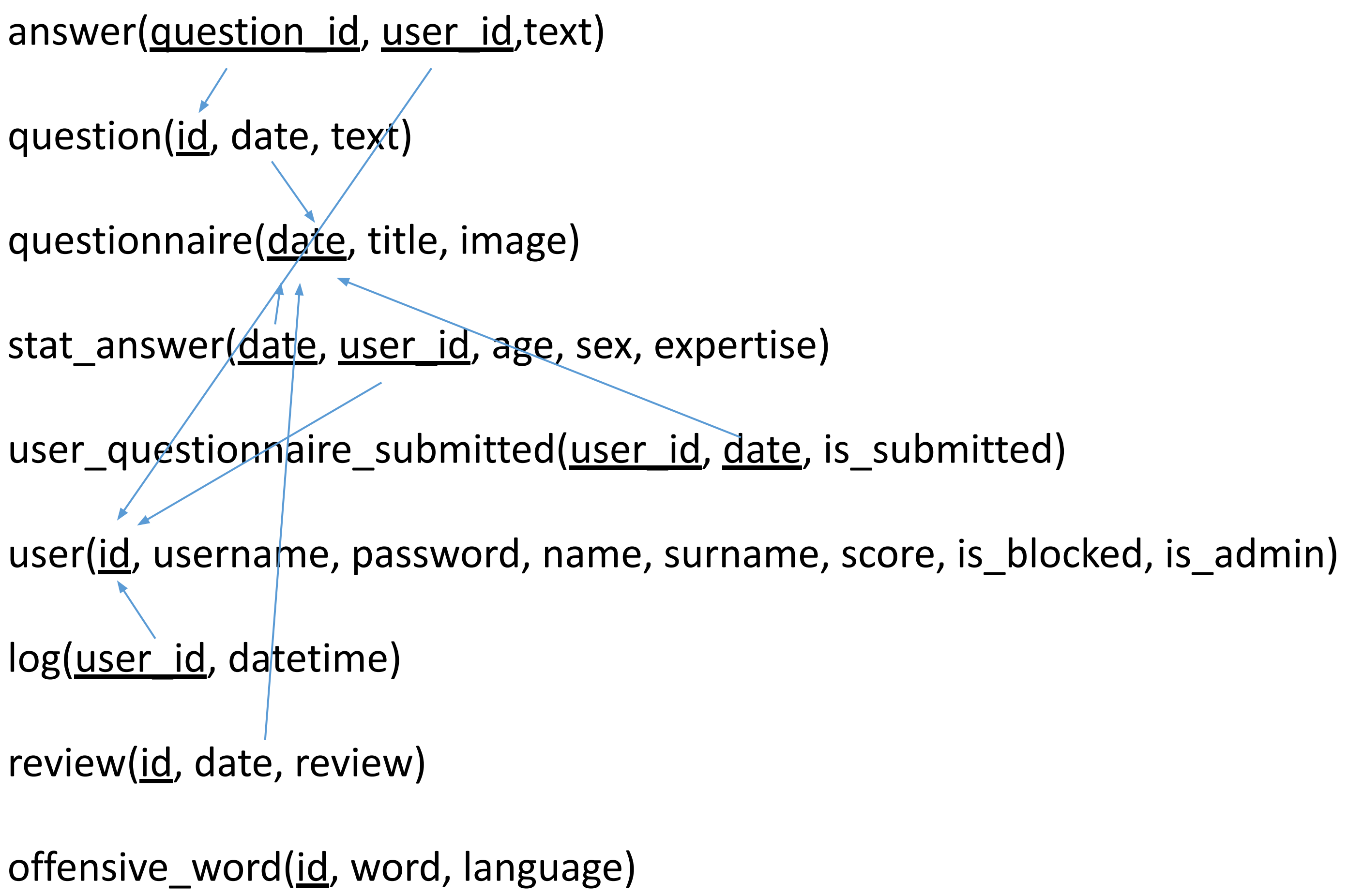
E-R diagram


review
id
date
text
1:1
quest_review
0:N
stat_answer
age
sex
expertise
0:N
0:N
id
username
password
name
surname
score
is_blocked
is_admin
user
questionnaire
date
title
image
submitted_quest
0:N
0:N
0:N
0:N
1:N
question_in_quest
is_submitted
answer
1:1
0:N
text
question
id
date
text
user_log
1:1
log_id
user_id
datetime
log
id
word
language
offensive_word

Motivations

stat_answer and submitted_quest are represented as different relations as they reflect two different logical roles in the application and consequently their implementation in JPA.

Logical model



answer(question_id, user_id,text)

question(id, date, text)

questionnaire(date, title, image)

stat_answer(date, user_id, age, sex, expertise)

user_questionnaire_submitted(user_id, date, is_submitted)

user(id, username, password, name, surname, score, is_blocked, is_admin)

log(user_id, datetime)

review(id, date, review)

offensive_word(id, word, language)

Trigger Motivations

Triggers are used to update the score field in the user table every time a marketing or statistical answer is added to or removed from the database. This avoids expensive queries calculating the current score every time the leaderboard is shown.

A trigger is used to cascade the deletion of the questionnaire onto answer and stat_answer, as the delete performed by the foreign key constraint does not activate triggers in MySQL

Trigger

```
delimiter $$
CREATE TRIGGER questionnaire_cleanup
BEFORE DELETE ON questionnaire FOR EACH ROW
BEGIN
     DELETE FROM answer WHERE answer.question_id in
          (SELECT question.id FROM question WHERE question.date = old.date);
     DELETE FROM stat_answer WHERE stat_answer.date = old.date;
END $$
delimiter ;
```

Trigger

```
CREATE TRIGGER update_score_on_answer
AFTER INSERT ON answer FOR EACH ROW
UPDATE user SET user.score = user.score + 1
WHERE new.user_id = user.id

CREATE TRIGGER decrement_on_delete_answer
AFTER DELETE ON answer FOR EACH ROW
UPDATE user
SET user.score = user.score - 1
WHERE old.user_id = user.id
```

Trigger

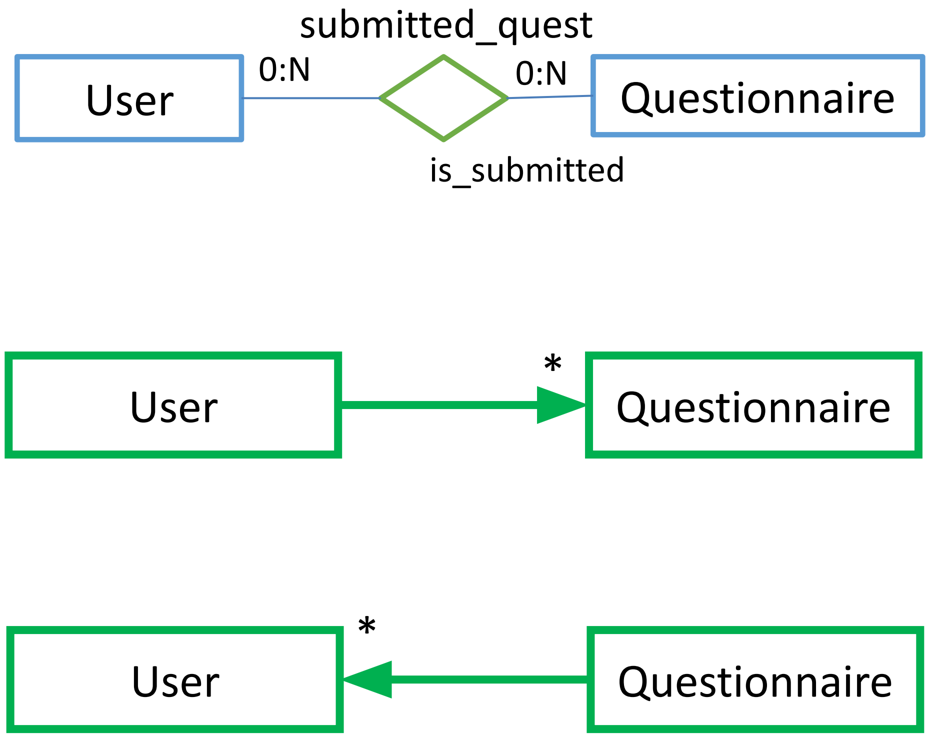
```
CREATE TRIGGER update_score_on_statistics
AFTER INSERT ON stat_answer for each row
UPDATE user
SET user.score = (user.score +
    2*(IF((new.age) IS NOT NULL, 1, 0))
    +
    2*(IF((new.sex) IS NOT NULL, 1, 0))
    +
    2*(IF((new.expertise) IS NOT NULL, 1, 0)))
WHERE user.id = new.user_id;
```

This trigger uses the mysql IF() function, which returns 1 if the condition evaluates to true, 0 otherwise.

Trigger

```
CREATE TRIGGER decrement_on_delete_stat_answer
AFTER DELETE ON stat_answer FOR EACH ROW
UPDATE user
SET user.score = user.score - (
    2*(IF((old.age) IS NOT NULL, 1, 0))
    +
    2*(IF((old.sex) IS NOT NULL, 1, 0))
    +
    2*(IF((old.expertise) IS NOT NULL, 1, 0)))
WHERE user.id = old.user_id;
```

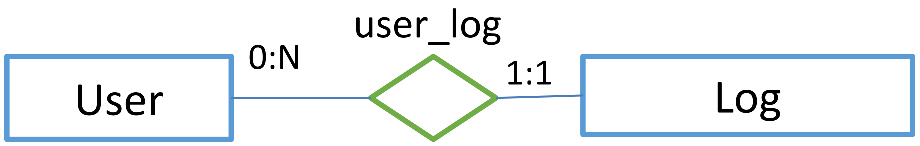
Relationship “submitted_quest”



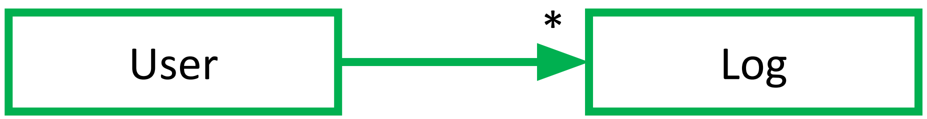
- User -> Questionnaire @ElementCollection is used, since there is a boolean attribute on the relation, which says whether the user submitted the questionnaire or not.
  It is implemented as a map<Integer, Boolean> in User, where the key is the Questionnaire id

- Questionnaire -> User is implemented through queries for performance reasons.
  The queries can easily retrieve all users who have is_submitted set to true or false

Relationship “user_log”

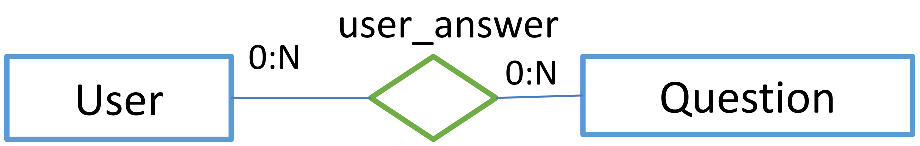


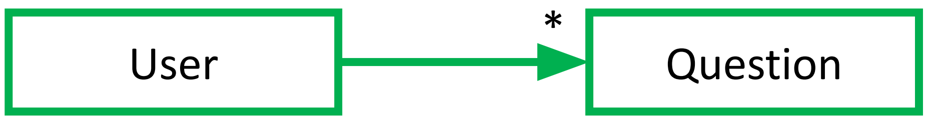
- User -> Log
  @ElementCollection as the logs are dependent from the User, and only have one attribute (datetime)



- Log -> User
  not needed

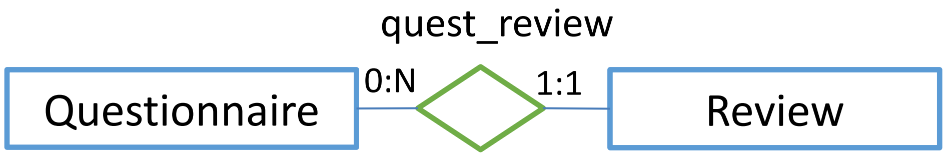
Relationship “user_answer”

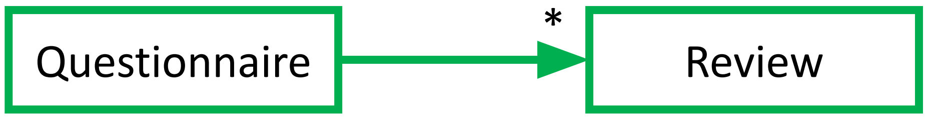




- User -> Question
  @ElementCollection is used to obtain the answer given a user and a question.
  It is implemented as a Map<Integer, String> where the key is the question id. We deemed an Answer entity not necessary as its only significant attribute is a simple String.
- Question -> User
  not needed

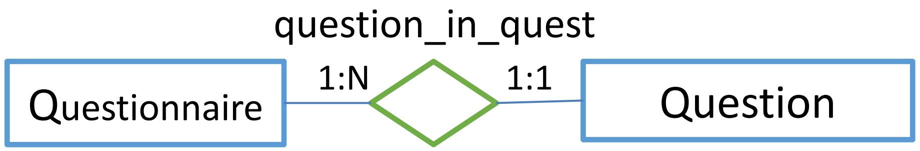
Relationship “quest_review”



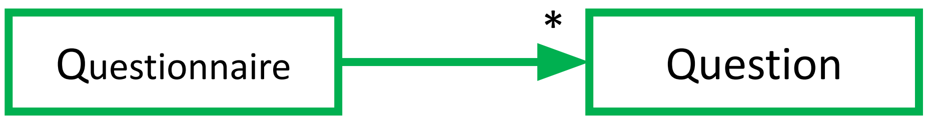


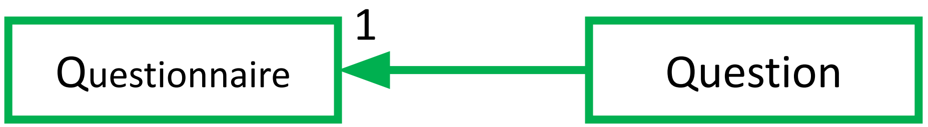
- Questionnaire -> Review @ElementCollection used to avoid creating a Review entity, because reviews are strictly dependent on questionnaires and only have a text attribute
- Review -> Questionnaire not needed

Relationship “question_in_quest”



- Questionnaire->Question @OneToMany(mappedBy=”questionnaire”) is necessary to get the questions of a questionnaire



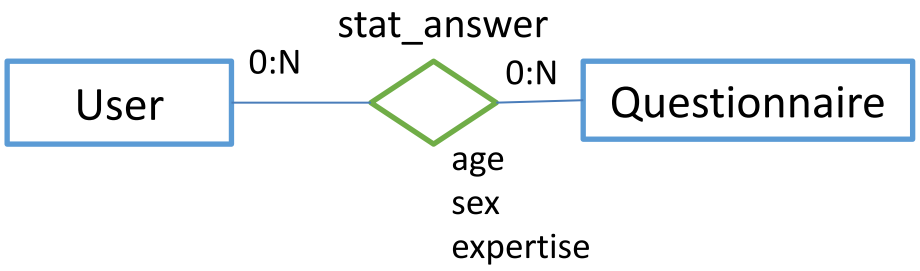


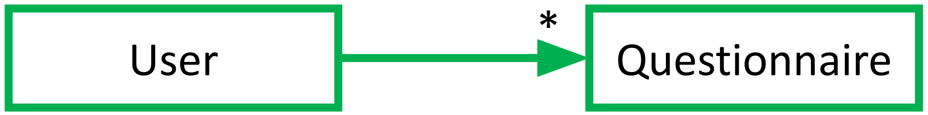
- Question-> Questionnaire @ManyToOne, retrieve the questionnaire given a question

  @JoinColumn(name = "date", referencedColumnName = "date")

  This annotation is used to do a join with an attribute different from the primary key

Relationship “stat_answer”





- User -> Questionnaire

  @ElementCollection is needed to obtain the stat answers related to a questionnaire given a user.
  We need a StatAnswer entity because the relationship between User and Questionnaire has more than one attribute, hence a single Map<Date, attribute> would have not sufficed. In fact we mapped the relation with a Map<Date, StatAnswer> in User, where Date is the primary key of Questionnaire and StatAnswer is associate to that User and Questionnaire.

- Questionnaire -> User not needed

Entity User - Named Queries

```
@NamedQueries({
@NamedQuery(name = "User.getUsersWhoSubmitted", query =
     "SELECT u FROM User u JOIN u.isSubmitted s WHERE KEY(s) = ?1 AND s = true"),
@NamedQuery(name = "User.getUsersWhoCanceled", query =
     "SELECT u FROM User u JOIN u.isSubmitted s WHERE KEY(s) = ?1 AND s = false"),
@NamedQuery(name = "User.checkCredentials", query =
     "SELECT r FROM User r  WHERE r.username = ?1 and r.password = ?2"),
@NamedQuery(name = "User.getNonAdminUsers", query =
     "SELECT r FROM User r WHERE r.is_admin = FALSE ORDER BY r.score DESC"),
@NamedQuery(name = "User.checkUnique", query =
     "SELECT r FROM User r WHERE r.username = ?1 or r.email = ?2")})
```

MOTIVATIONS

- User.checkCredentials: used for credentials verification
- User.getNonAdminUsers: allows to retrieve non admin users to be shown in the leaderboard.
- User.getUsersWhoSubmitted: used to show users who completed and submitted the questionnaire to the admin
- User.getUsersWhoCanceled: sed to show users who canceled the questionnaire to the admin
- User.checkUnique: used during registration process to check whether the inserted username or mail are already associated to an existing account

Entity User

```
public class User implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    private String password;

    private String surname;

    private String username;

    private boolean is_admin;

    private boolean is_blocked;

    private String email;

    private Integer score;
```

User Relationships

```
@ElementCollection
@CollectionTable(name = "user_questionnaire_submitted", schema = "db_marketing",
joinColumns = @JoinColumn(name = "user_id", referencedColumnName = "id"))
@MapKeyColumn(name = "date")
@Column(name = "is_submitted")
private Map<Date, Boolean> isSubmitted;

@ElementCollection(fetch=FetchType.LAZY)
@CollectionTable(name = "log", schema = "db_marketing", joinColumns =
@JoinColumn(name = "user_id", referencedColumnName = "id"))
@Column(name = "datetime")
private Set<Date> logs;

@ElementCollection
@CollectionTable(name = "answer", schema = "db_marketing", joinColumns =
@JoinColumn(name = "user_id", referencedColumnName = "id"))
@MapKeyColumn(name = "question_id")
@Column(name = "answer")
private Map<Integer, String> answers;

@OneToMany(mappedBy="user")
@MapKey(name="date")
private Map<Date, StatAnswers> statAnswers;
```

Entity Questionnaire

```
public class Questionnaire implements Serializable {

    @Id
    @Temporal(TemporalType.DATE)
    private Date date;

    private String title;

    @Basic(fetch = FetchType.LAZY)
    @Lob
    @Column(columnDefinition = "LONGBLOB")
    private byte[] image;

    @OneToMany(mappedBy = "questionnaire")
    private List<Question> questions;

    @ElementCollection
    @CollectionTable(name="review", schema="db_marketing",
joinColumns=@JoinColumn(name="date"))
    @Column(name = "review")
    private Set<String> reviews;

     ...
}
```

The image is lazily fetched for performance reasons

Entity Question

```
public class Question implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String text;

    @ManyToOne
    @JoinColumn(name = "date", referencedColumnName = "date")
    private Questionnaire questionnaire;

    ...
}
```

Entity StatAnswer

```
public class StatAnswer implements Serializable {

    @Id
    @Temporal(TemporalType.DATE)
    private Date date;

    @Id
    private Integer user_id;

    private Integer age;

    private String sex;

    private String expertise;

    @ManyToOne
    @JoinColumn(name="user_id", updatable = false, insertable = false)
    private User user;
    ...
}
```

Entity OffensiveWord

```
@NamedQueries({
@NamedQuery(name = "OffensiveWord.containsWord",
        query = "SELECT COUNT(o) FROM OffensiveWord o WHERE ?1
                                        LIKE CONCAT('%',o.word,'%')")
})
public class OffensiveWord implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String word;

    private String language;

    ...
}
```

This entity is necessary to perform the query that checks whether the user input contains an offensive word. In fact JPQL requires us to have the entity to access its word field.

We used an open source database for the offensive words table.
Credits: https://github.com/turalus/encycloDB

Motivations

All relations have no cascade properties, as all cascade deletes are handled through foreign key constraints or triggers in the database.

Business tier components

- @Stateless UserService
  - `User checkCredentials(String username, String pwd)`
  - `User getUser(int userid)`
  - `boolean checkUnique(String usrn, String mail)`
  - `List<user> getNonAdminUsersRefreshed()`
  - `void addUser(String usrn, String name, String surname, String pwd, String mail)`
  - `void registerAccess(User u)`
  - `void blockUser(User user)`

- @Stateless QuestionnaireOperationsService
  - `List<User> getUsersWhoSubmitted(Date date)`
  - `List<User> getUsersWhoCanceled(Date date)`
  - `boolean checkNotStartedNorFinished`
  - `void beginQuestionnaire(User u, Questionnaire qst)`
  - `void submitQuestionnaire(User u, Questionnaire qst)`
  - `boolean isSubmitted(User u, Questionnaire qst)`
  - `void addReview(String text, Questionnaire qst)`
  - `Set<String> getAllReviews(Questionnaire q)`

Business tier components

- @Stateless QuestionnaireManagerService
  - `public List<Questionnaire> getAll()`
  - `public void createQuestionnaire(ArrayList<String> questions, Date date, String title, byte[] imageData)`
  - `public void deleteQuestionnaire(Date deletionDate)`
  - `public boolean questionnaireAlreadyExist(Date plannedDate)`
  - `public Questionnaire findByDate(Date date)`
  - `public Questionnaire getToday()`
  - `public void addAnswers(User u, Map<Integer, String> answers)`
  - `public void addStatAnswers(Questionnaire q, User u, Integer age, String sex, String expertise)`
  - `public List<String> getAnswersToQuestions(User u, List<Question> questions)`
  - `public boolean containsOffensiveWords(Collection<String> answers)`
  - `public List<Question> getQuestions(Questionnaire q)`
  - `public byte[] getQuestionnaireImage(Questionnaire q)`
  - `public StatAnswers getStatAnswers(User user, Questionnaire questionnaire)`

Motivations

All EJBs are stateless, as all requests are served independently and interact with the database. The only state kept between calls is the **partial answers** given by the user and not yet committed to the database. These are kept in the session for easier retrieval in case the user changes his mind and wants to submit the questionnaire.
Furthermore, answers are encoded as a Map<Integer, String>, thus only using basic Java types. A stateful EJB would have added useless complexity without any real advantage, since we don't need any of the features provided by an extended persistence context.

UserServices contains all business methods related to User retrieval, login, and checks

QuestionnaireManagerServices handles the creation, deletion, and retrieval of questionnaires

QuestionnaireOperationsServices handles everything related to submitting questionnaire and retrieving data related to past submissions

Servlets

There are several servlets organized by user and admin that provide different services:

Common

- Check Login
- Register
- Logout
- ServletBase*

User

- Home
- TodaysQuestionnaire: gets the questionnaire of the day
- TodaysQuestionnaireStatistics: displays statistical questions
- Leaderboard: allows to show the Leaderboard to the user

Admin

- AdminHome: shows the admin home with the main functionalities he can access
- CreateQuestionnaire: allows to create the main structure of the questionnaire
- QuestionnaireDetails: allows to create specific questions
- AdminDelete: used to delete a questionnaire
- AdminHistory: allows to see previously created questionnaires
- GetAnswers: retrieve answer of user to a specific questionnaire

*The ServletBase class is extended by all other servlets and provides utility functions to check if the user is logged in, redirections and thymeleaf template generation

Authors

Luigi Fusco

Francesco Fulco Gonzales

Alberto Latino