Gamified Application

Arianna Galzerano - 10563365

Leonardo Giusti - 10633778

Francesco Govigli - 10556637

POLITECNICO
MILANO 1863

Specifications

The goal of this project is to develop a web application to collect data from registered users about their feedbacks on products which are made available by administrators.

The project can be divided in 2 main parts:

User App:

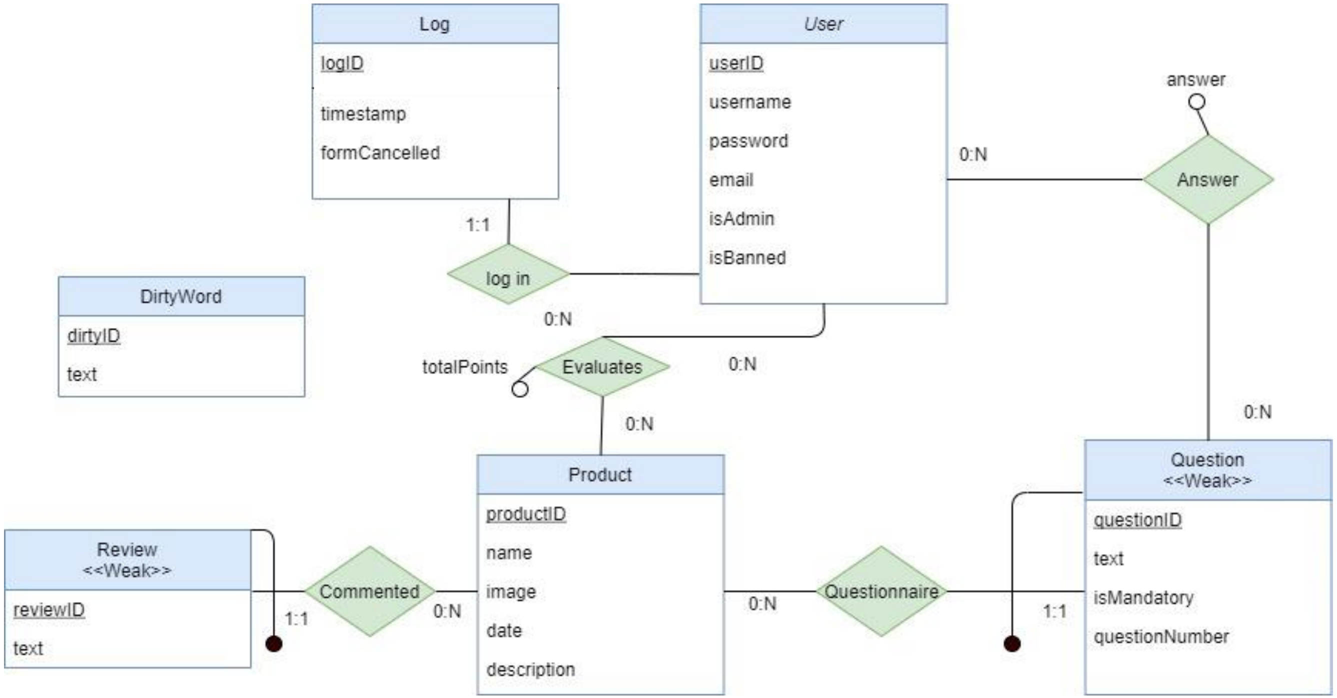
- Main functionalities are the compilation of the proposed questionnaire and the visualization of a leaderboard composed by all users who filled in the form for the specific product.

  Scoring is done following a specific policy explained in the next slides.

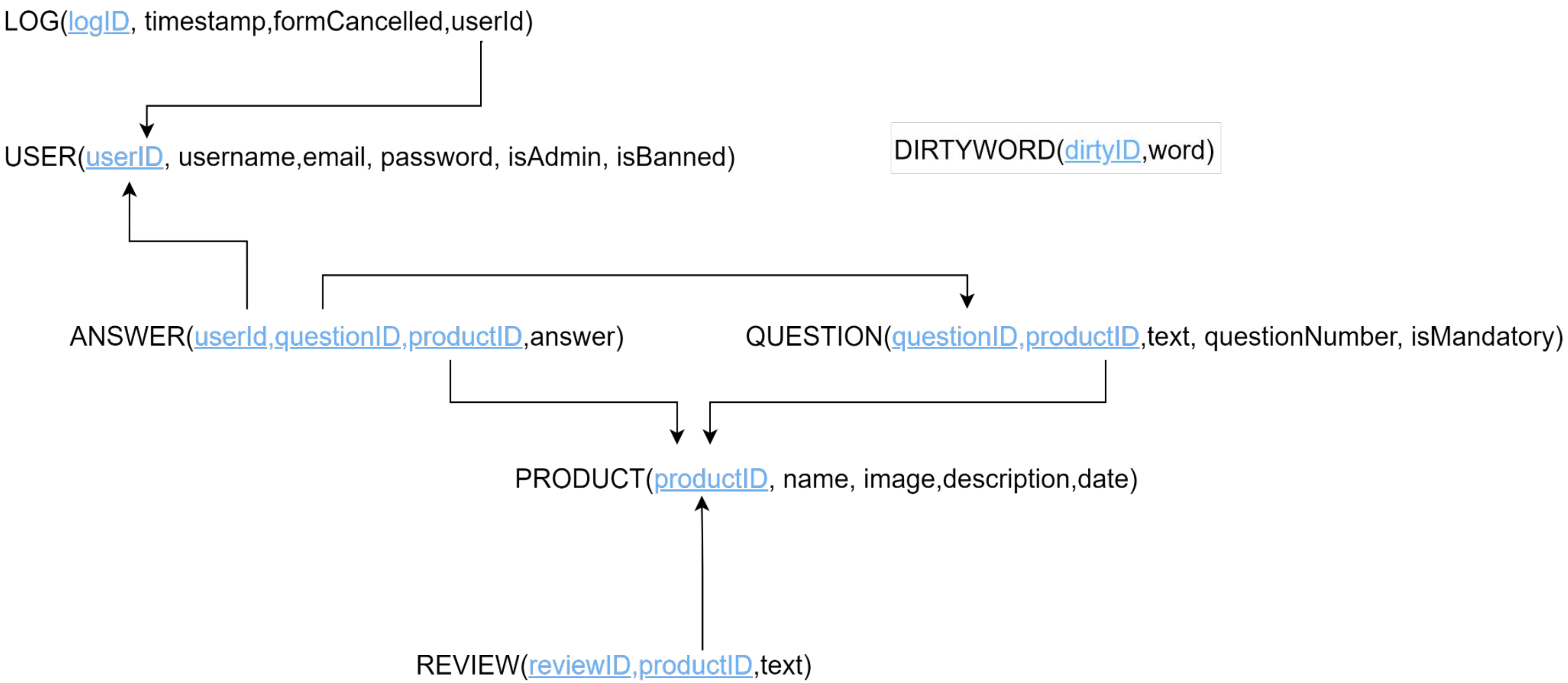
Admin App:

- Main functionalities are the creation of new products for a present or posterior date and the inspection and deletion of past data.

Database ER-Schema



Logical Schema



LOG(logID, timestamp,formCancelled,userId)

USER(userID, username,email, password, isAdmin, isBanned)

DIRTYWORD(dirtyID,word)

ANSWER(userId,questionID,productID,answer)

QUESTION(questionID,productID,text, questionNumber, isMandatory)

PRODUCT(productID, name, image,description,date)

REVIEW(reviewID,productID,text)

Design choices

- In the admin application there was no need of the deletionPage and we added just the delete functionality of an inspected questionnaire.

- Reviews are dynamically inserted, only by the admin, during the creation of a product.

- Admins are assumed to be already registered in the application and don’t need to Signup

- Review and Question entities are identified as weak w.r.t Product because they have no meaning with no associated product.

- The user ban is managed only in a soft-way, where the user can still access, but cannot fill in the questionnaire

- The deletion is managed by deleting the product corresponding to the specified date and thus, all the associated entities.

- The information that a user has canceled a questionnaire is saved in the user log

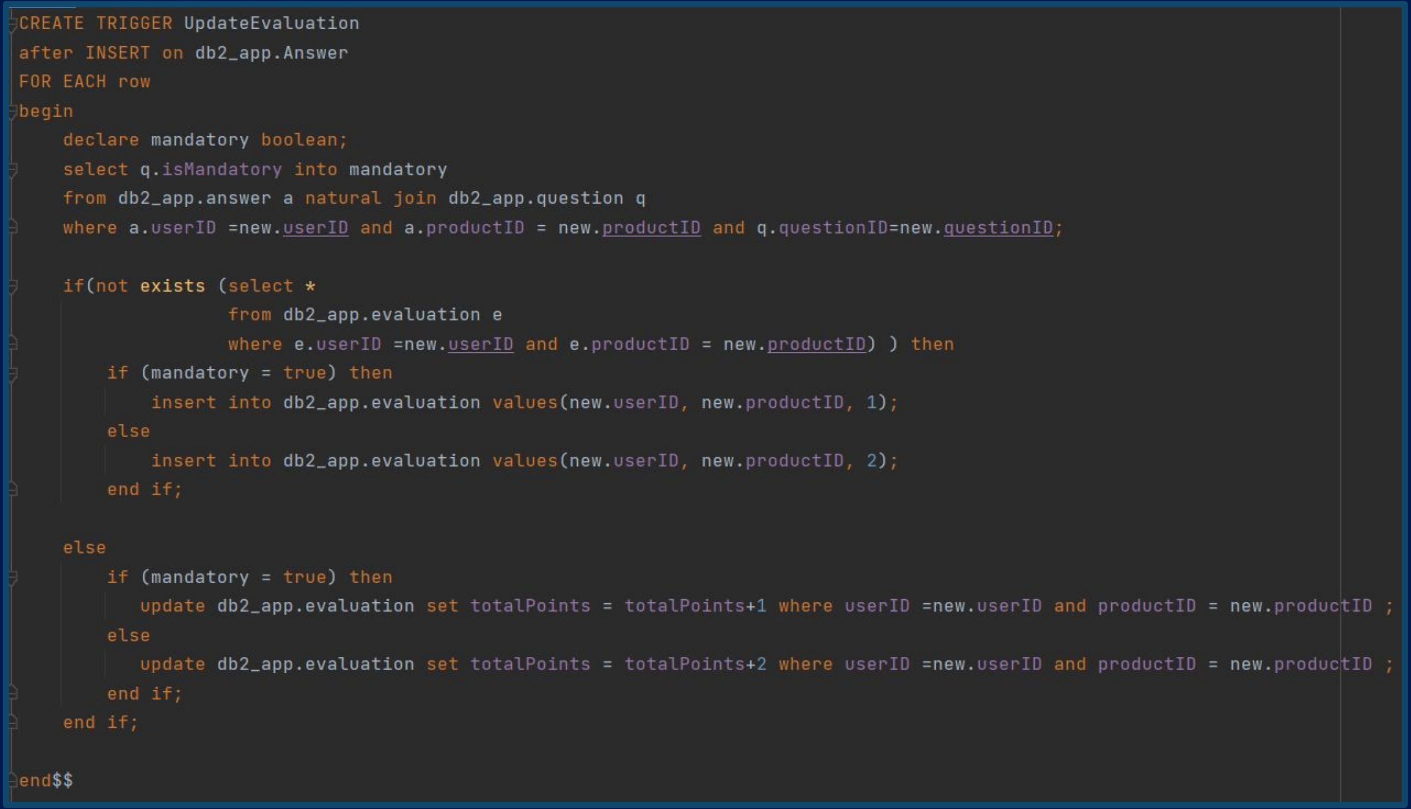
Trigger’s Motivation

As part of the implementation, a Trigger is inserted in order to automatically compute the total points a User obtains by submitting the answers to a questionnaire.

Every time a mandatory or optional answer is inserted into the ANSWERS table, the points gained with the new answer are added to the total score for a specific user and for the product of the day in the EVALUATION table.

This choice was made in order to avoid expensive queries to compute the total points that have to be displayed in the LEADERBOARD.

Trigger Implementation



```
CREATE TRIGGER UpdateEvaluation
after INSERT on db2_app.Answer
FOR EACH row
begin
    declare mandatory boolean;
    select q.isMandatory into mandatory
    from db2_app.answer a natural join db2_app.question q
    where a.userID =new.userID and a.productID = new.productID and q.questionID=new.questionID;

    if(not exists (select *
                   from db2_app.evaluation e
                   where e.userID =new.userID and e.productID = new.productID) ) then
        if (mandatory = true) then
            insert into db2_app.evaluation values(new.userID, new.productID, 1);
        else
            insert into db2_app.evaluation values(new.userID, new.productID, 2);
        end if;

    else
        if (mandatory = true) then
           update db2_app.evaluation set totalPoints = totalPoints+1 where userID =new.userID and productID = new.productID ;
        else
           update db2_app.evaluation set totalPoints = totalPoints+2 where userID =new.userID and productID = new.productID ;
        end if;
    end if;

end$$
```

Evaluation Policy:

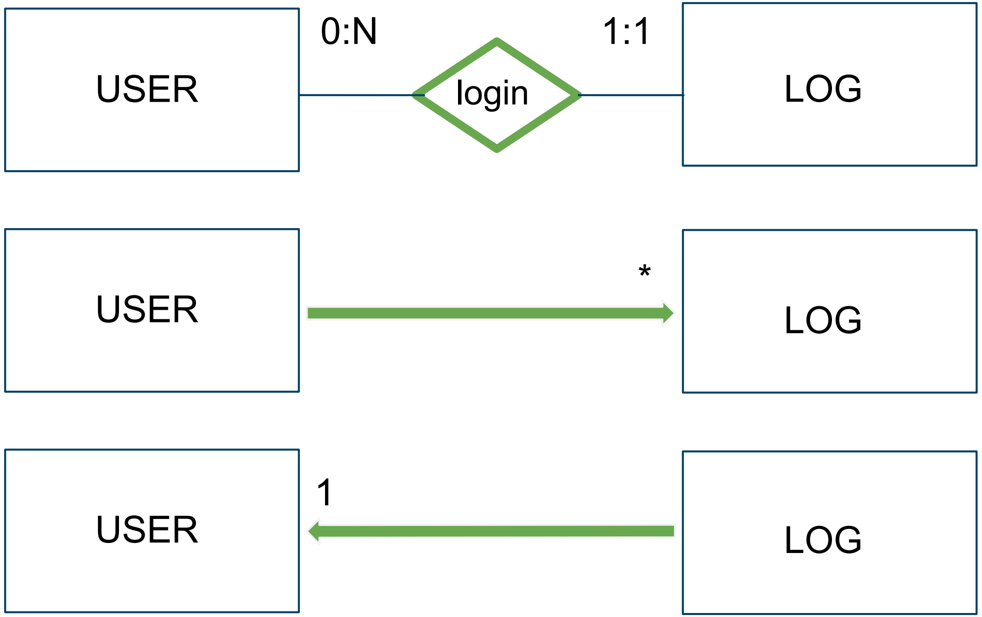
Through the trigger, the rules applied for the total scores computations consist in:

- +1 for replying to a mandatory question
- +2 for replying to a optional question

Idea:

It creates a new evaluation if there are still none related to the specific user and product, else
it just updates the totalPoints associated to the instance

Relationship “log in”



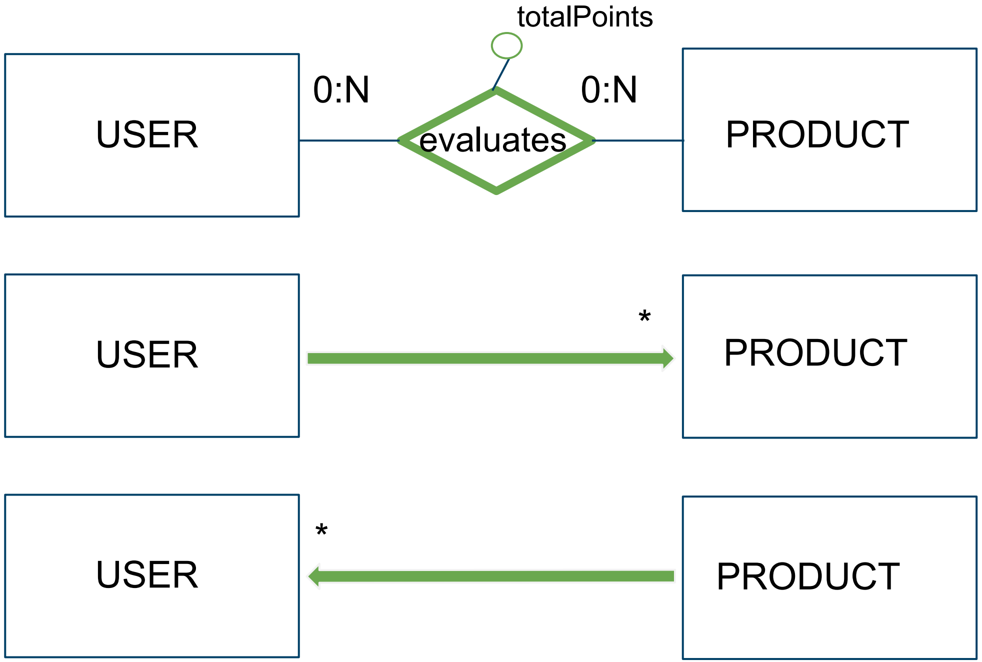
- User -> Log (@OneToMany)

  We just consider it bidirectional for simplicity, even if it is not needed by the specifications.

- Log -> User (@ManytoOne)

  This relationship is used to retrieve the current log instance of a user when he decides to cancel a questionnaire to set the log attribute isFormCancelled = true.

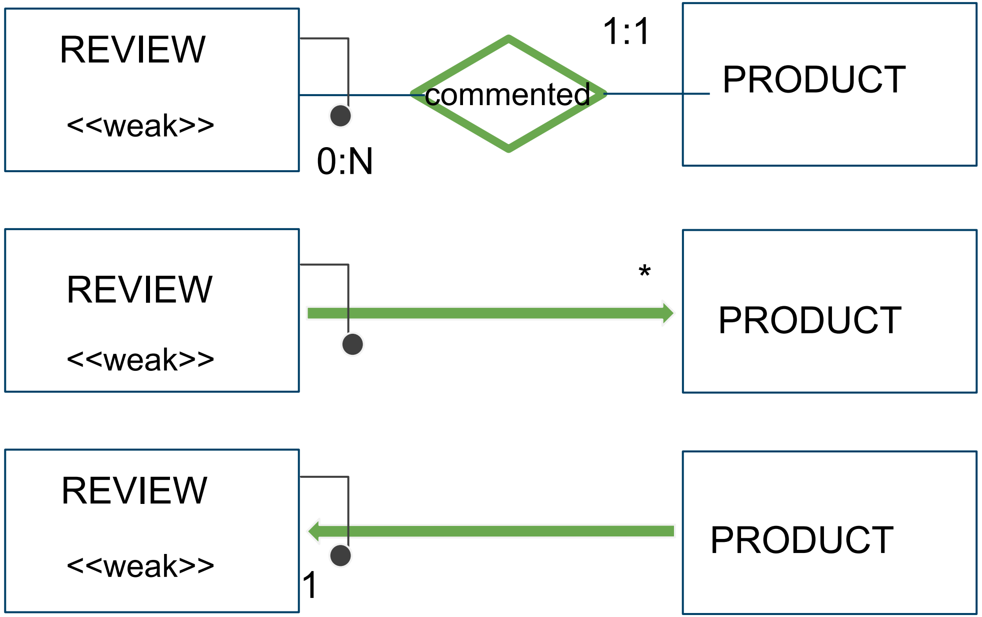
Relationship “evaluates”



For this relationship we have created a JoinTable called Evaluation, which contains the 2 collections of both users and products with a @ManyToMany bidirectional relationship.

Evaluation Table is just a utility to manage and keep the totalPoints of a user updated in order to create the Leaderboard.

Relationship “commented”



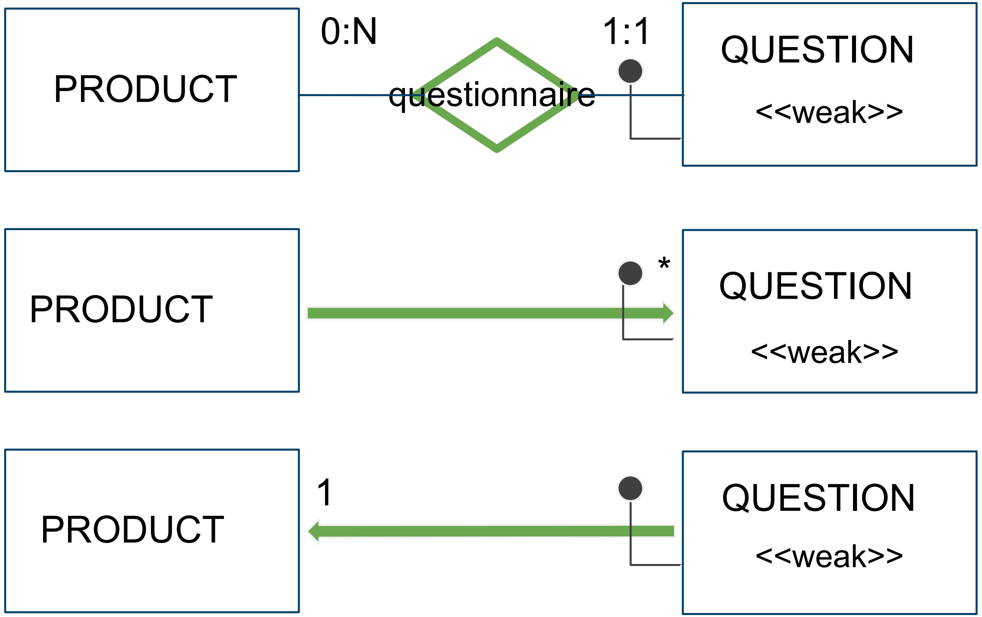
- Product->Review (@OneToMany)

  This relationship is used to retrieve the reviews of a given product which can be added only by the admins on the application

- Review->Product (@ManytoOne)

  We just consider it bidirectional for simplicity even if it is not needed by the specifications.

Relationship “questionnaire”



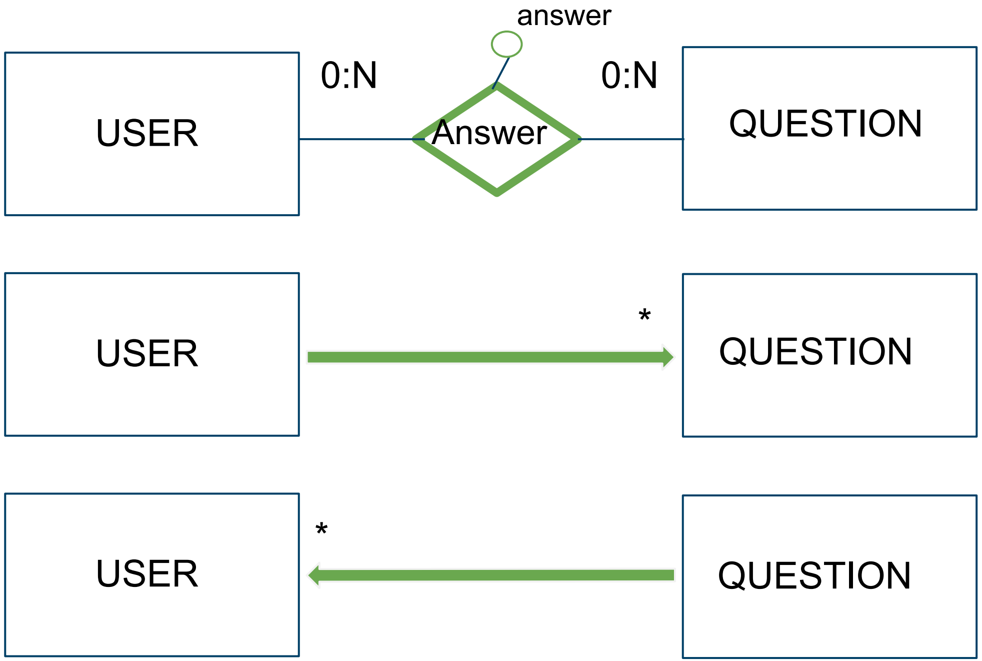
- <u>Product->Question(@OneToMany)</u>

This relationship is necessary to retrieve all the questions for the product of the day, to display them in the questionnaire for the users.

- <u>Question->Product (@ManytoOne)</u>

This side of the relationship is kept for simplicity and for completion purpose.

Relationship “Answer”



With this relationship N:N between Question and User we are able to track the answers that a User gives to a question (related to a specific product since question is a weak entity).

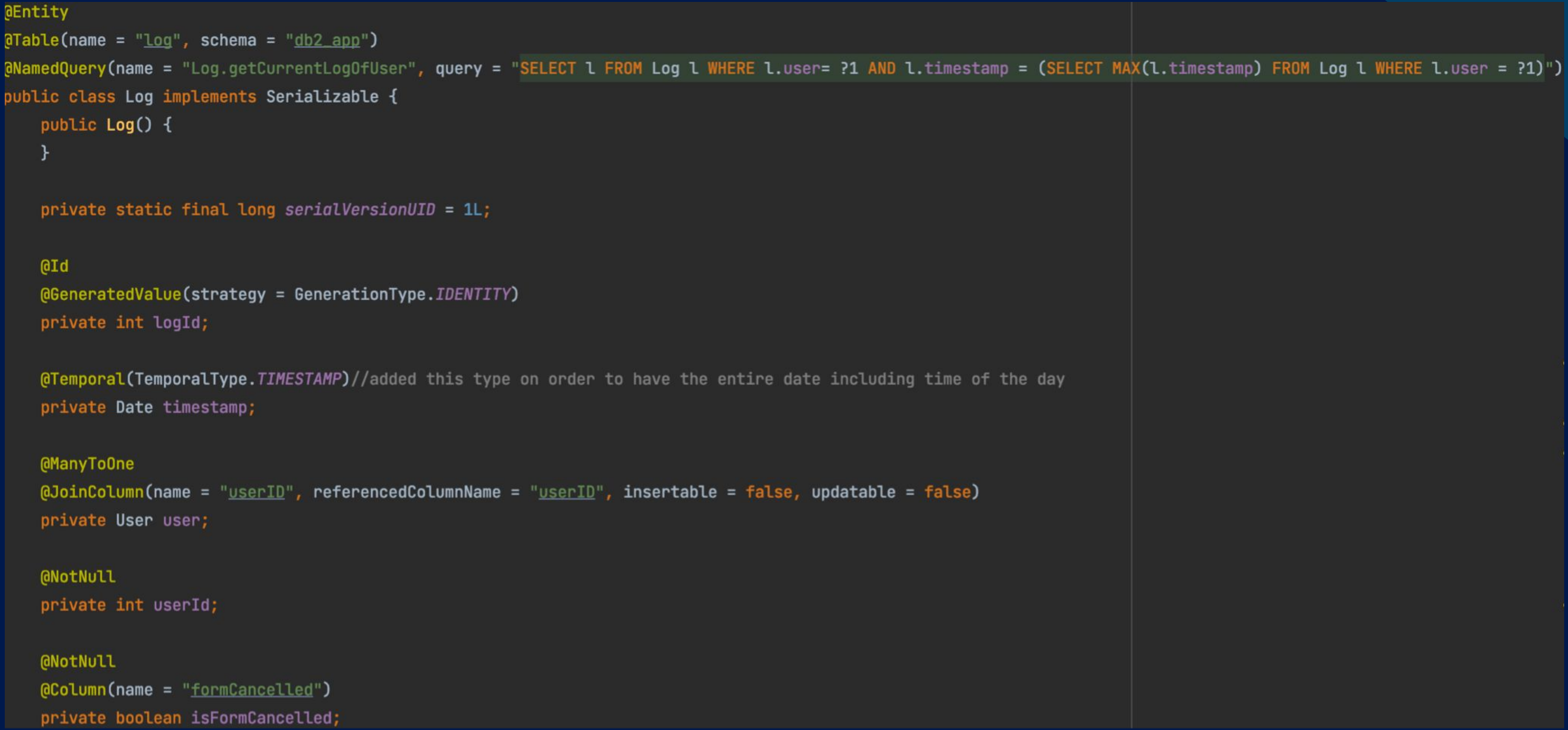
In the project, it is modelled as a new entity Answer which has two unidirectional @ManyToOne relationships with Question and with User because for each *user* there are *multiple answers* and for each *question* there are *multiple answers*.

We decided to model unidirectionally the relationship because in the inspection the author of an answer is needed and also the related question.

Entities design

- User
- Question
- Review
- Product
- Log
- Dirty Word

Log Entity



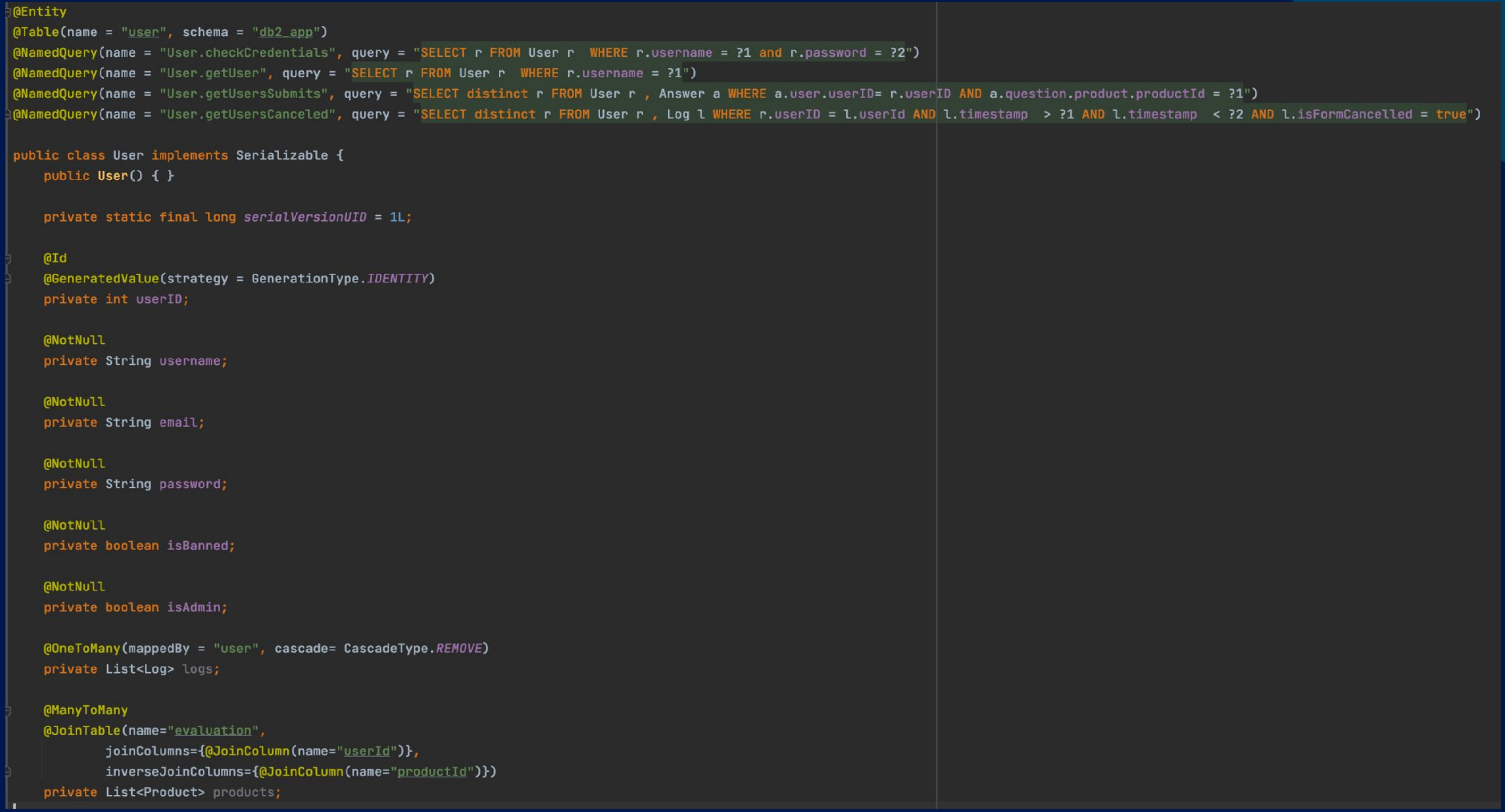
```
@Entity
@Table(name = "log", schema = "db2_app")
@NamedQuery(name = "Log.getCurrentLogOfUser", query = "SELECT l FROM Log l WHERE l.user= ?1 AND l.timestamp = (SELECT MAX(l.timestamp) FROM Log l WHERE l.user = ?1)")
public class Log implements Serializable {
    public Log() {
    }

    private static final long serialVersionUID = 1L;

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

    @Temporal(TemporalType.TIMESTAMP)//added this type on order to have the entire date including time of the day
    private Date timestamp;

    @ManyToOne
    @JoinColumn(name = "userID", referencedColumnName = "userID", insertable = false, updatable = false)
    private User user;

    @NotNull
    private int userId;

    @NotNull
    @Column(name = "formCancelled")
    private boolean isFormCancelled;
```

Motivations

- *isFormCancelled* introduced to note down questionnaire canceled by users because we do not save the record in the Answers table;
  The reason behind this design choice is that it is given the chance for a user to first cancel and then submit a questionnaire in the same day

- Bidirectional many-to-one association Log to User :
  - Owner class is Log
  - *insertable* = false, *updatable* = false because it's not the responsibility of the Log entity to create or update a User

User Entity

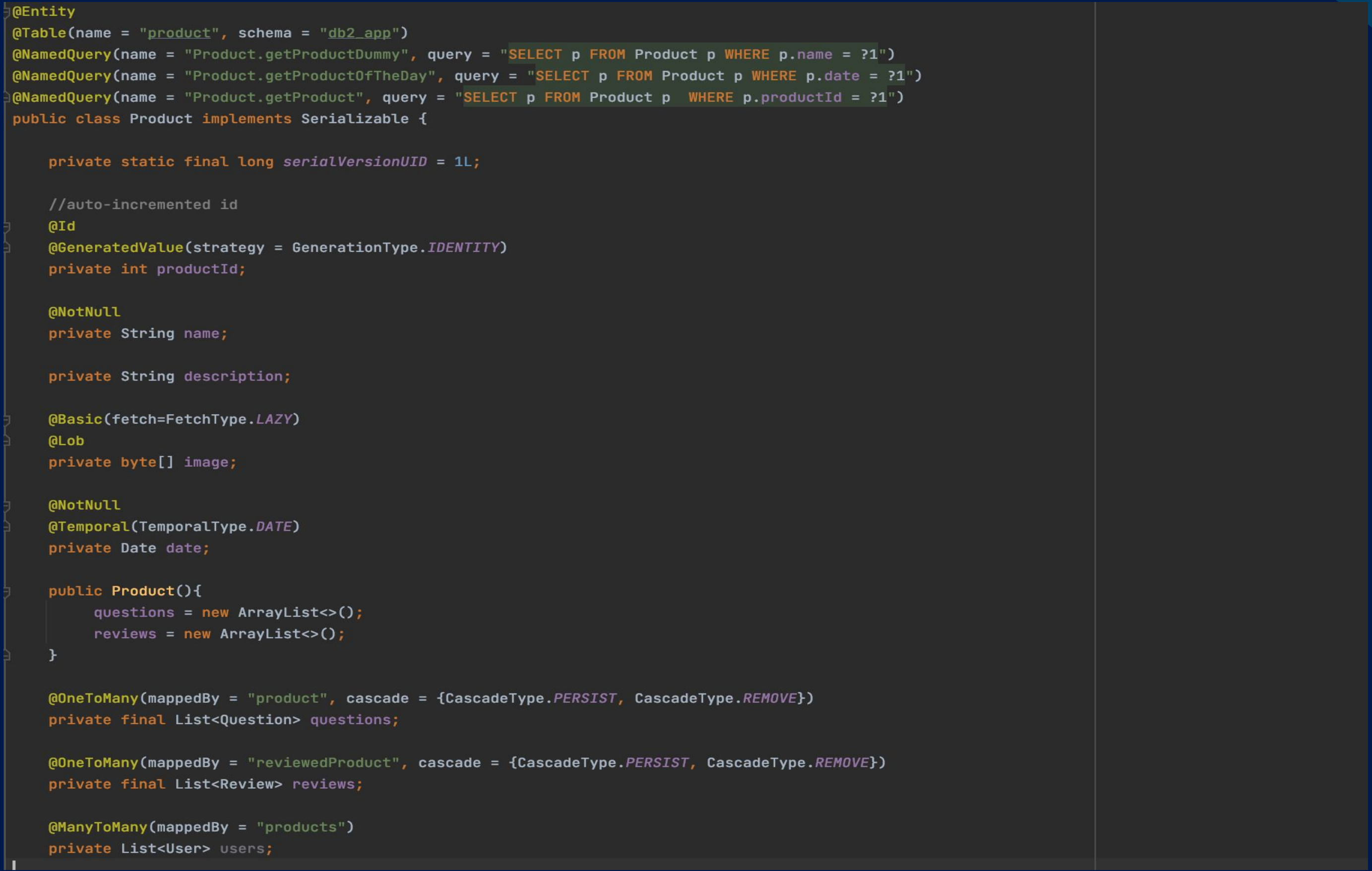


```java
@Entity
@Table(name = "user", schema = "db2_app")
@NamedQuery(name = "User.checkCredentials", query = "SELECT r FROM User r  WHERE r.username = ?1 and r.password = ?2")
@NamedQuery(name = "User.getUser", query = "SELECT r FROM User r  WHERE r.username = ?1")
@NamedQuery(name = "User.getUsersSubmits", query = "SELECT distinct r FROM User r , Answer a WHERE a.user.userID= r.userID AND a.question.product.productId = ?1")
@NamedQuery(name = "User.getUsersCanceled", query = "SELECT distinct r FROM User r , Log l WHERE r.userID = l.userId AND l.timestamp  > ?1 AND l.timestamp  < ?2 AND l.isFormCancelled = true")

public class User implements Serializable {
    public User() { }

    private static final long serialVersionUID = 1L;

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

    @NotNull
    private String username;

    @NotNull
    private String email;

    @NotNull
    private String password;

    @NotNull
    private boolean isBanned;

    @NotNull
    private boolean isAdmin;

    @OneToMany(mappedBy = "user", cascade= CascadeType.REMOVE)
    private List<Log> logs;

    @ManyToMany
    @JoinTable(name="evaluation",
            joinColumns={@JoinColumn(name="userId")},
            inverseJoinColumns={@JoinColumn(name="productId")})
    private List<Product> products;
```

Motivations

- Bidirectional Many-to-Many association Product to User :
  - JoinTable is Evaluation that has as primary keys userID and productID
- Bidirectional One-to-Many association Log to User :
  - Owner class is Log
- User.getUsersCanceled is a Named Query used to retrieve all the logs of a given user from 00:00.01 to 23:59.59 of a certain day

Product Entity



```
@Entity
@Table(name = "product", schema = "db2_app")
@NamedQuery(name = "Product.getProductDummy", query = "SELECT p FROM Product p WHERE p.name = ?1")
@NamedQuery(name = "Product.getProductOfTheDay", query = "SELECT p FROM Product p WHERE p.date = ?1")
@NamedQuery(name = "Product.getProduct", query = "SELECT p FROM Product p  WHERE p.productId = ?1")
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    //auto-incremented id
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int productId;

    @NotNull
    private String name;

    private String description;

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

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

    public Product(){
         questions = new ArrayList<>();
         reviews = new ArrayList<>();
    }

    @OneToMany(mappedBy = "product", cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private final List<Question> questions;

    @OneToMany(mappedBy = "reviewedProduct", cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private final List<Review> reviews;

    @ManyToMany(mappedBy = "products")
    private List<User> users;
```

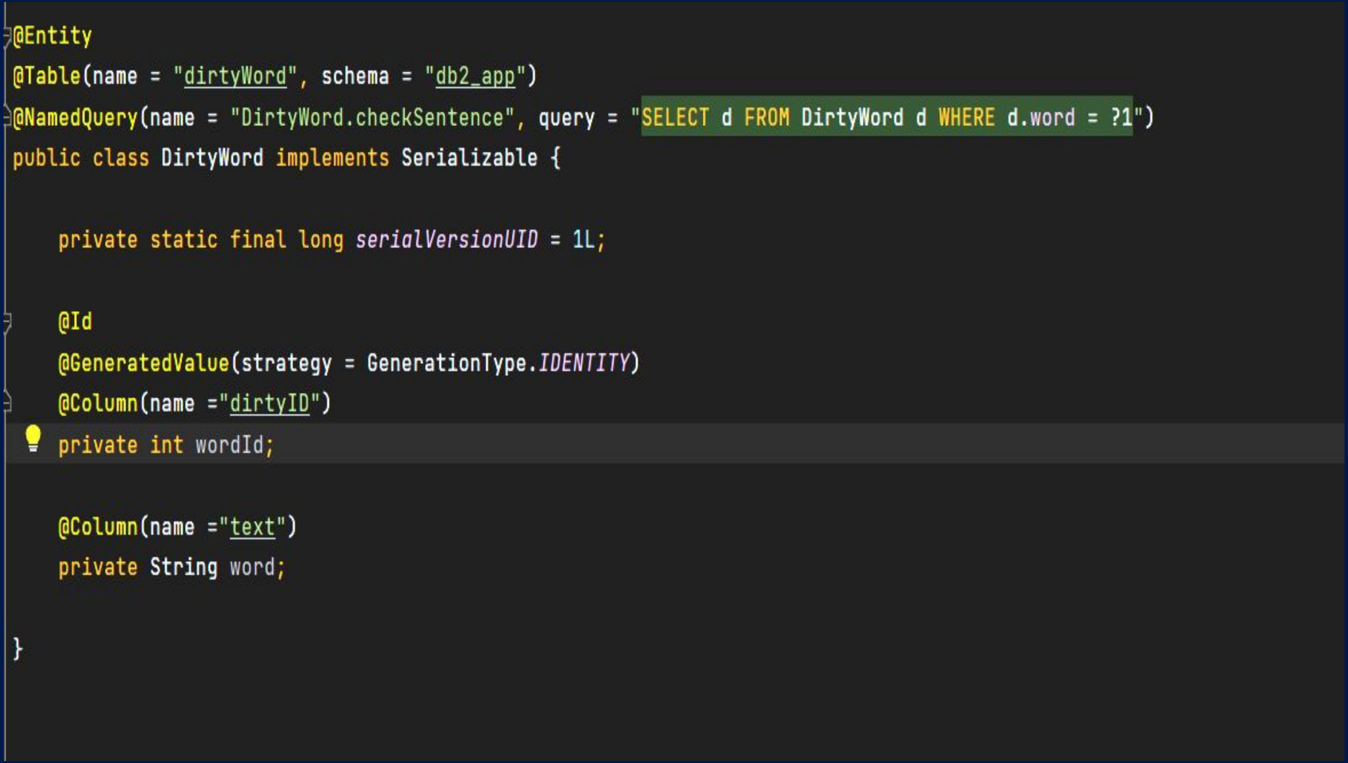
Motivations

- Since the product related questions and reviews are two weak entities, the CascadeType is set to:
  CascadeType.PERSIST → to persist all questions and reviews of a specific item
  CascadeType.REMOVE → to directly delete the instances when the product is deleted by the admin in the inspection page.

- *OrphanRemoval deletes an instance if it is no more attached to any parent in any relationship. Since we are just adding the attached reviews and questions just during the product creation phase and never dynamically changing the relationship collection, we decided to use the cascading instead of OrphanRemoval*

- FetchType is kept to EAGER, since every time we access the product we are interested in getting questions.

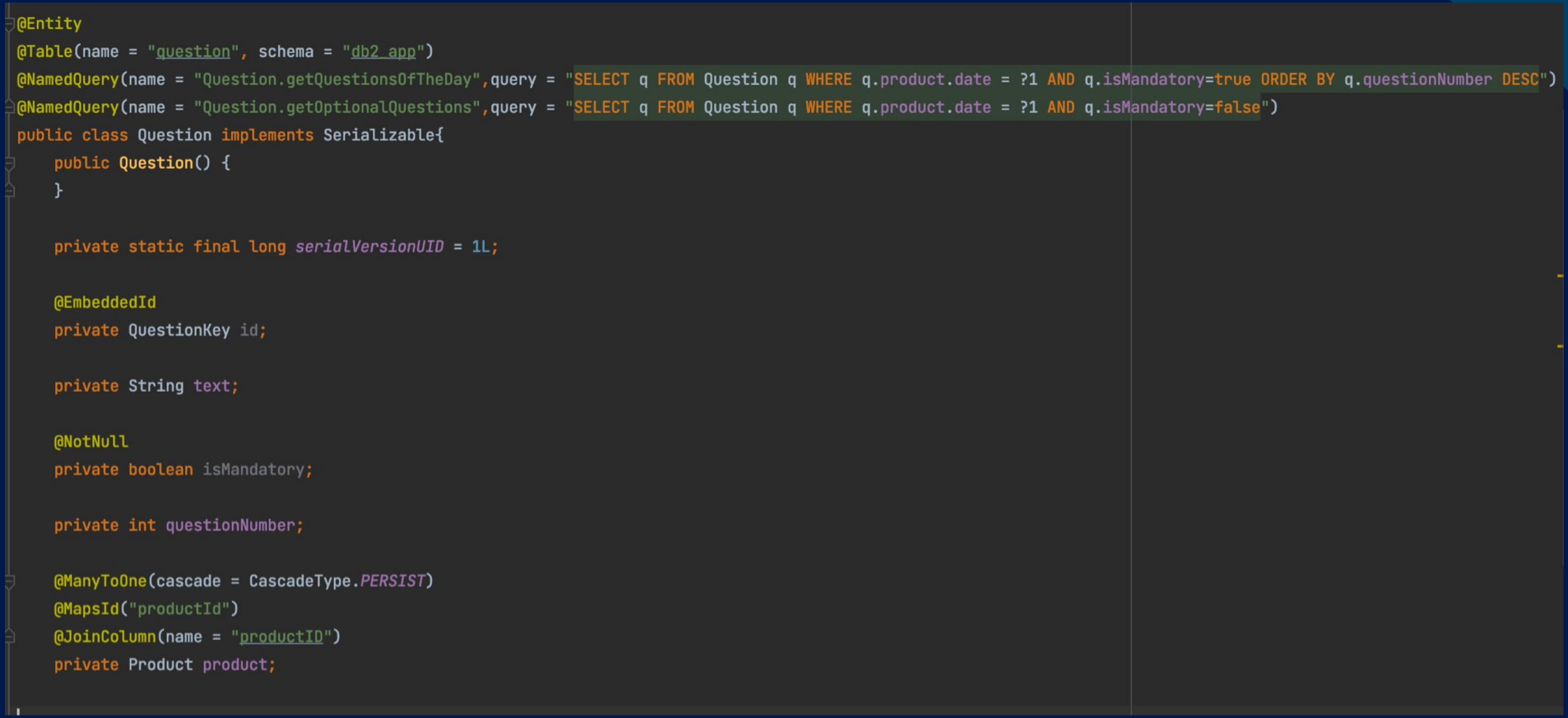
Dirty Word Entity



```
@Entity
@Table(name = "dirtyWord", schema = "db2_app")
@NamedQuery(name = "DirtyWord.checkSentence", query = "SELECT d FROM DirtyWord d WHERE d.word = ?1")
public class DirtyWord implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name ="dirtyID")
    private int wordId;

    @Column(name ="text")
    private String word;

}
```

Dirty words are directly stored in the DB.
They are kept independent w.r.t other entities since the words are static and fixed instances.

Question Entity

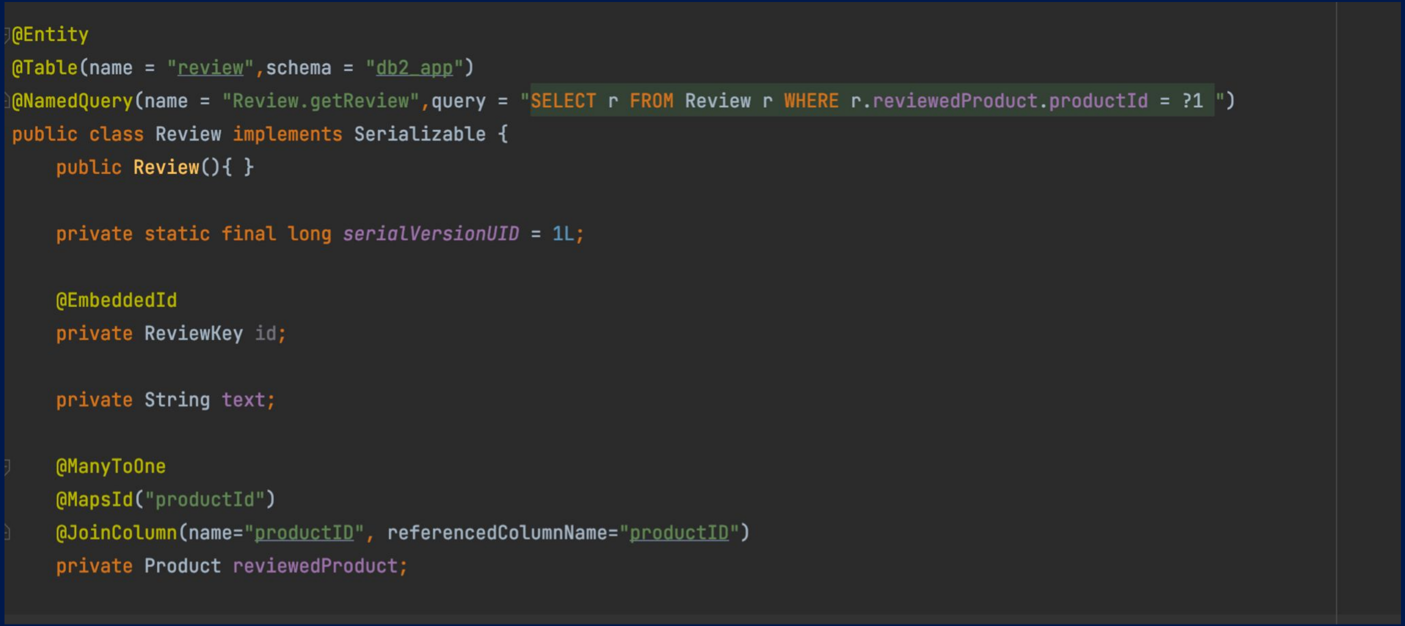


```
@Entity
@Table(name = "question", schema = "db2_app")
@NamedQuery(name = "Question.getQuestionsOfTheDay",query = "SELECT q FROM Question q WHERE q.product.date = ?1 AND q.isMandatory=true ORDER BY q.questionNumber DESC")
@NamedQuery(name = "Question.getOptionalQuestions",query = "SELECT q FROM Question q WHERE q.product.date = ?1 AND q.isMandatory=false")
public class Question implements Serializable{
    public Question() {
    }

    private static final long serialVersionUID = 1L;

    @EmbeddedId
    private QuestionKey id;

    private String text;

    @NotNull
    private boolean isMandatory;

    private int questionNumber;

    @ManyToOne(cascade = CascadeType.PERSIST)
    @MapsId("productId")
    @JoinColumn(name = "productID")
    private Product product;
```

Motivations

- Since a questionnaire exists only if related to a Product, the Question entity is weak and depends on the Product entity
- Named query ‘*Question.getQuestionsOfTheDay*’ can be used to extract the ordered collection of questions of a product by means of a JPQL query, as an alternative to relationship navigation
- Bi-directional @ManytoOne association Question to Product, where the former entity is defined as Owner
- Persistence operations are cascaded to the dependent entity Question

Review Entity

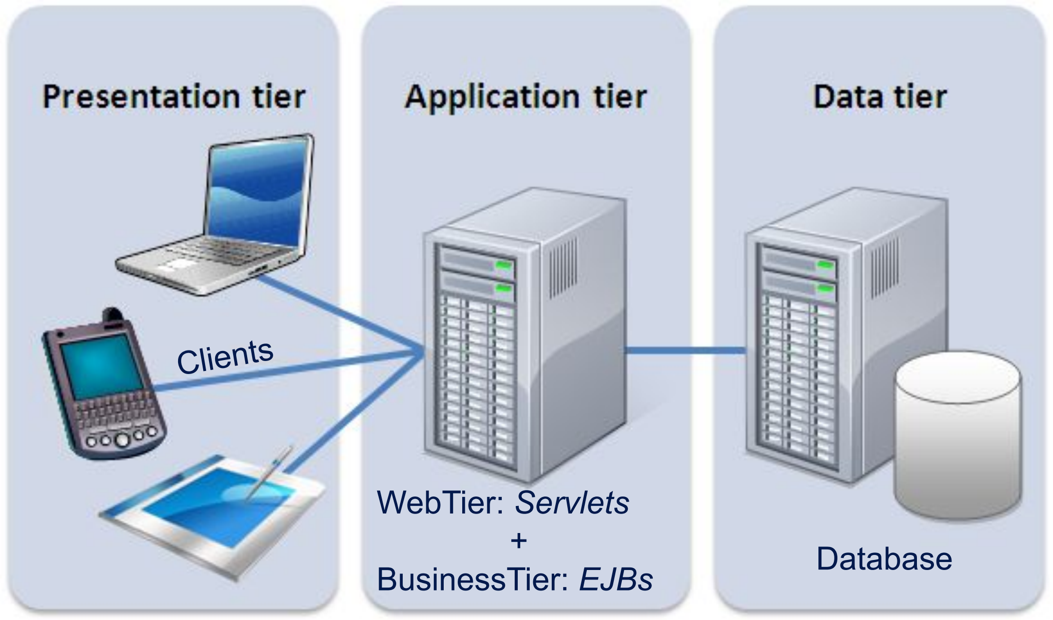


```
@Entity
@Table(name = "review",schema = "db2_app")
@NamedQuery(name = "Review.getReview",query = "SELECT r FROM Review r WHERE r.reviewedProduct.productId = ?1 ")
public class Review implements Serializable {
    public Review(){ }

    private static final long serialVersionUID = 1L;

    @EmbeddedId
    private ReviewKey id;

    private String text;

    @ManyToOne
    @MapsId("productId")
    @JoinColumn(name="productID", referencedColumnName="productID")
    private Product reviewedProduct;
```

Bi-directional one-to-one association to Product

Review is the owner entity

Components



Web Tier Components:

Login/Logout
Error

ADMIN side:
Creation - post new product
Deletion - post deleted product
Inspection - inspect previous inserted products

USER side:
GoToHomepage - get homepage required data through a json file
QuestionnaireData - get data for questionnaire through json file
Leaderboard - get data for Leaderboard through json
AnswerData - post users' answers

Business Tier Components - ProductService

@Stateless ProductService:

- `public Product getProductOfTheDay(Date date)`
- `public Product checkDateAvailability(Date date)`
- `public void createNewProduct(String name,String description,Date date,List<String> questions,byte[] image,List<String> productReviews)`
- `public static byte[] readImage(InputStream imageInputStream)`
- `public void deleteProduct(Product product)`

Business Tier Components -

UserService

@Stateless UserService:

- `public User addUser(String username, String email, String password, boolean banned)`
- `public User getUser(String username)`
- `public User checkCredentials(String username, String password)`
- `public void banUser(String username)`
- `public UserStatus checkUserStatus(User user, Product product)`
- `public void LogUser(User user)`
- `public void cancelForm(User user)`
- `public List<String> getUsersWhoCanceled(Product product)`
- `public List<String> getUsersWhoSubmits(Product product)`
- `public List<String> getAnsweredQuestions(Product product,String username)`
- `public Date getCorrectFormatDate(LocalDateTime now)`

Business Tier Components - QuestionnaireService

@Stateless QuestionnaireService:

- `public List<Question> getQuestionsOfTheDay()`
- `public List<Question> getOptionalQuestions()`
- `public List<String> convertToString(List<Question> questions)`

Business Tier Components -

AnswerService

@Stateless AnswerService:

- `public boolean hasDirtyWord(String[] answers)`
- `public String[] correctAnswerFormat(String[] answers)`
- `public void addAnswer(String response, User user, Question question)`
- `public boolean alreadyFilled(String username, Product product)`
- `public List<Answer> getUserAnswers(Product product, String s)`
- `public List<String> getAnswerText(List<Answer> answersFromUser)`
- `public boolean checkMandatoryOK(String[] mandatory_answers)`

Business Tier Components -

EvaluationService - ReviewService

@Stateless EvaluationService:

- `public List<Evaluation> getLeaderboard(Product product)`
- `public List<String> convertToString(List<Evaluation> evaluations)`

@Stateless ReviewService:

- `public ArrayList<String> getReviews(int productID)`

Motivations

Components are stateless because all client requests are served independently and update the database, no main memory conversational state need to be maintained

There is no need to save the states since all method calls are independent from each other and don’t interact between them

We could have done an alternative design by putting the ReviewService method directly inside ProductService