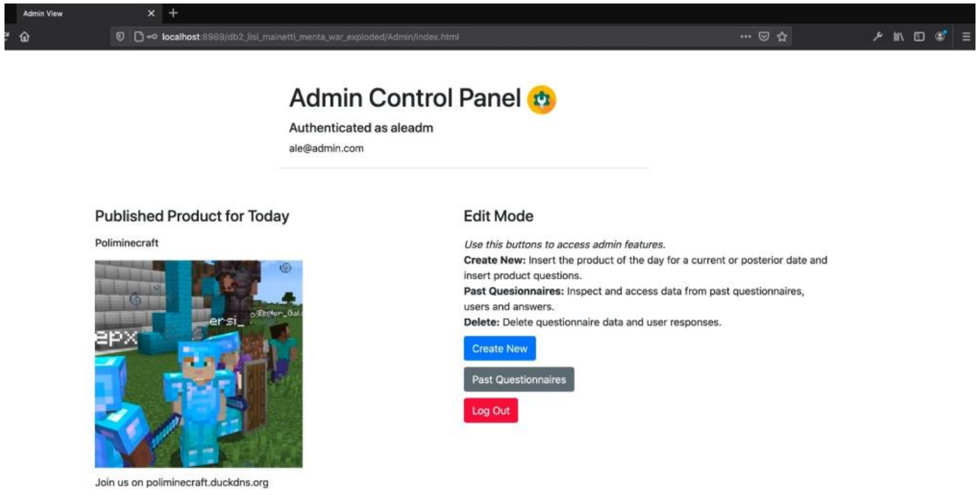

Admin View

localhost:8989/db2_lisi_mainetti_menta_war_exploded/Admin/index.html

Admin Control Panel

Authenticated as aleadm

ale@admin.com

Published Product for Today

Poliminecraft

Join us on poliminecraft.duckdns.org

Edit Mode

Use this buttons to access admin features.
Create New: Insert the product of the day for a current or posterior date and insert product questions.
Past Quesionnaires: Inspect and access data from past questionnaires, users and answers.
Delete: Delete questionnaire data and user responses.

Lisi Alessandro – Mainetti Lorenzo – Menta Andrea

Gamified Application

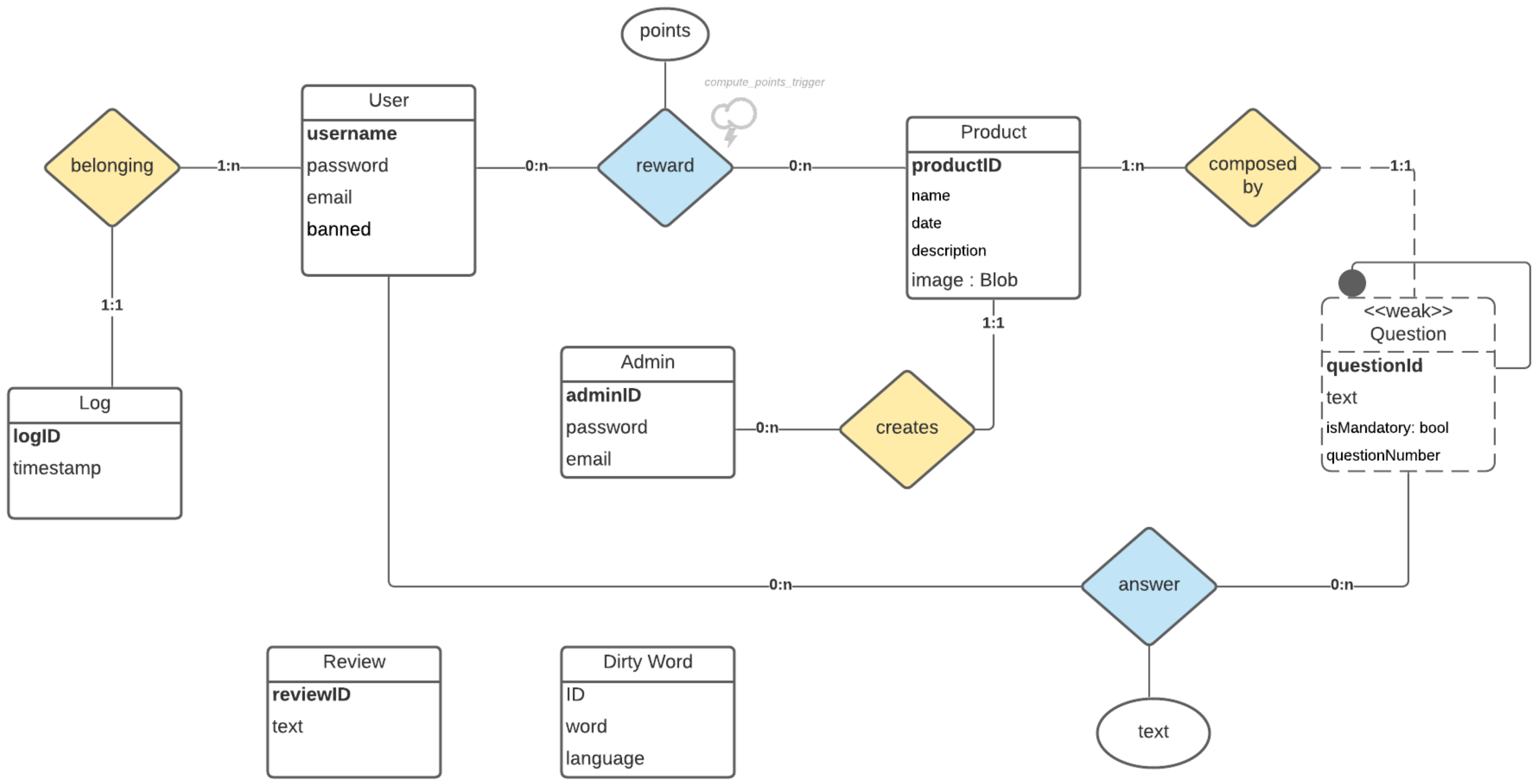
POLITECNICO
MILANO 1863

Specifications

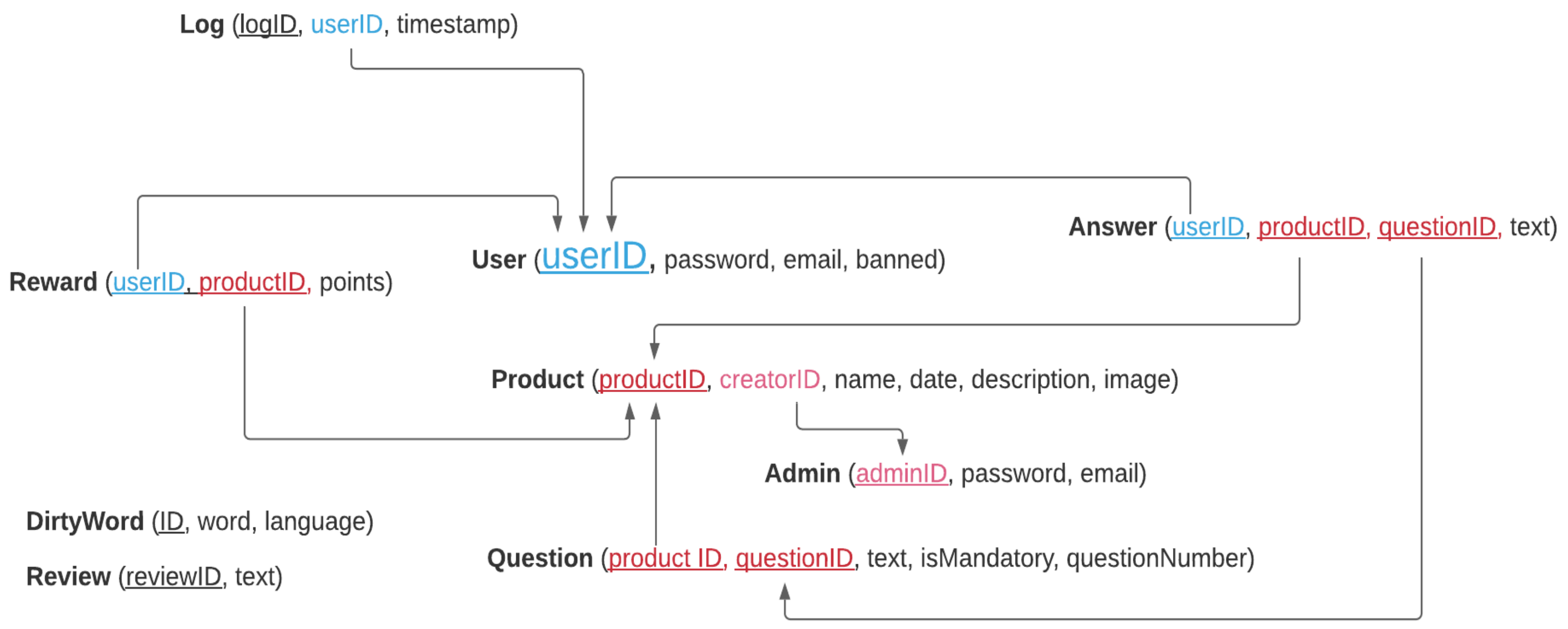
- The goal of the project is to implement a web app that deals with gamified consumer data collection.
- **User View**: a user can access the homepage where the product of the day is published, and he can complete the related questionnaire. The application computes the gamification points of each user that can be checked in the leaderboard.
- **Admin View**: an admin can access a reserved homepage where he is able to create a new questionnaire for the day of for a future date, inspect or delete a past questionnaire.

Gamified Marketing Web Application Database ER Design

Alessandro Lisi - Lorenzo Mainetti - Andrea Menta



Logical Schema



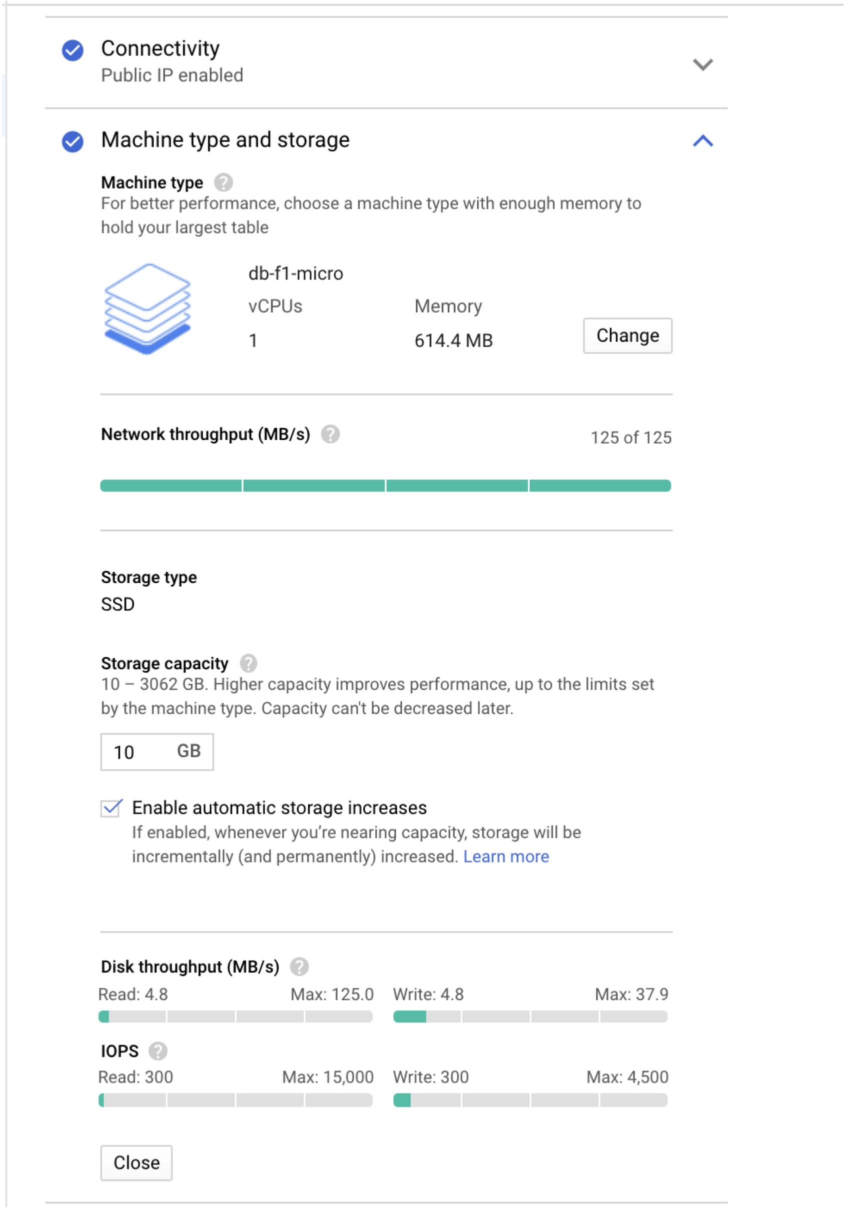
Log (logID, userID, timestamp)
User (userID, password, email, banned)
Answer (userID, productID, questionID, text)
Reward (userID, productID, points)
Product (productID, creatorID, name, date, description, image)
Admin (adminID, password, email)
DirtyWord (ID, word, language)
Review (reviewID, text)
Question (product ID, questionID, text, isMandatory, questionNumber)

Design

- A product cannot exist without a questionnaire, they are the same thing in the scope of the application. Given the absence of further specifications there would be no added value by modeling them as separate entities.
- Mandatory and optional questions are stored as the same entity, a boolean is used to differentiate them.
- An admin is not a common user and role "upgrade" is not supported by the application. Admins don't sign up, we assume they already have authentication credentials. This application is an example of RBAC.
- Different products can be created by different admin(s), therefore a product is associated with a unique creator identifier. Each admin has access only to his created products when it comes to inspection and deletion.
- Question is modeled as a weak entity because it cannot be identified without a product.
- Question number, which stores the order in which the admin inserted the question, is used (later on) to sort questions.
- Because the review submission by the user is not required by the specification, product reviews (!= user’s answers) are modeled as a table containing a set of random reviews.

MySQL Instance

- We have deployed a database instance using Google Cloud SQL. This way the database is shared between developers and always up to date.
- The machine type selected is intended for development and testing. It is not recommended for production use.
- Region: Zurich CH
- For better performance, we keep data close to the services that need it.



Connectivity
Public IP enabled

Machine type and storage

Machine type
For better performance, choose a machine type with enough memory to hold your largest table

db-f1-micro

vCPUs	Memory
1	614.4 MB

Change

Network throughput (MB/s) 125 of 125

Storage type
SSD

Storage capacity
10 – 3062 GB. Higher capacity improves performance, up to the limits set by the machine type. Capacity can't be decreased later.

10 GB

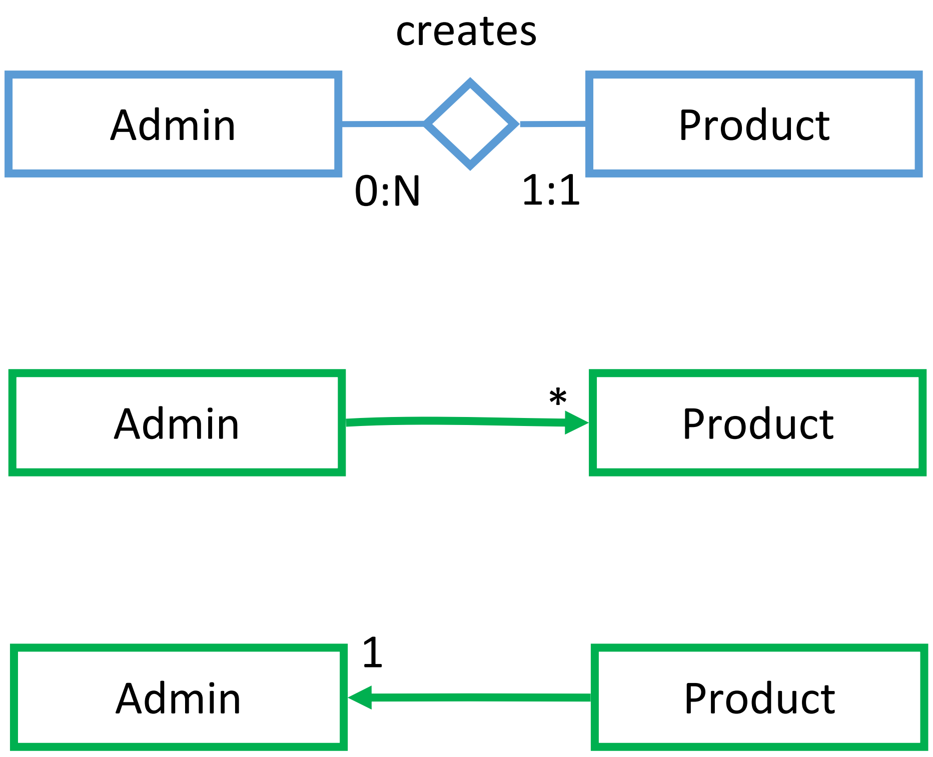
✓ Enable automatic storage increases
If enabled, whenever you're nearing capacity, storage will be incrementally (and permanently) increased. Learn more

Disk throughput (MB/s)
Read: 4.8 Max: 125.0 Write: 4.8 Max: 37.9

IOPS
Read: 300 Max: 15,000 Write: 300 Max: 4,500

Close

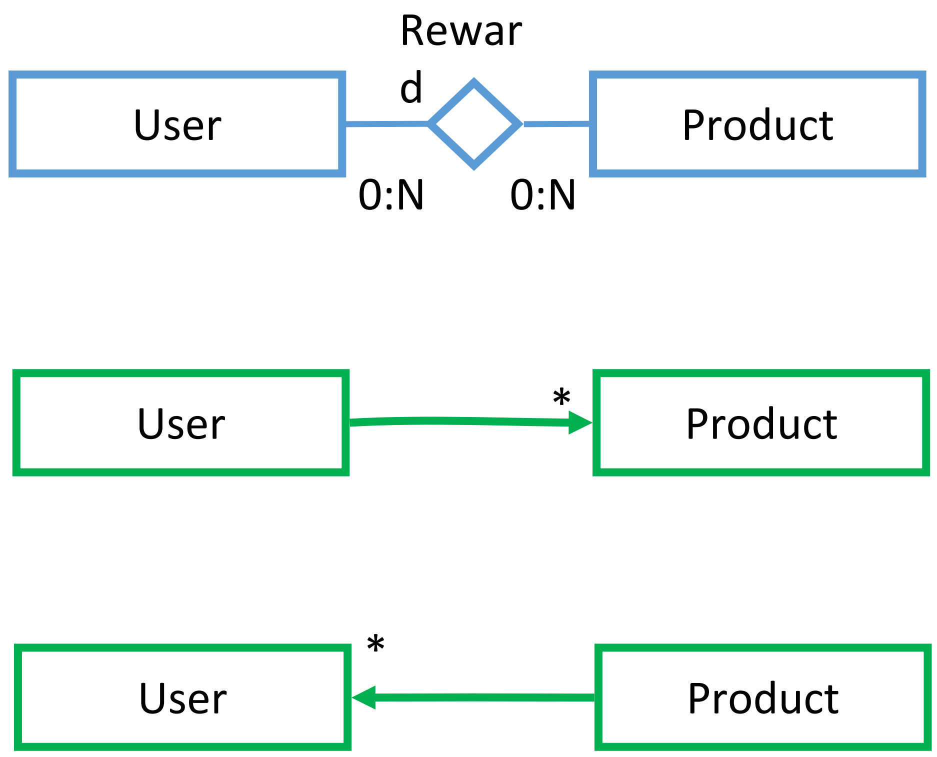
Relationship "creates"



- On Admin: @OneToMany
- Because an admin creates several products, and a product is not shared between different admins.
- On Product: @ManyToOne
- the other side of the relationship.

- Modeled as bidirectional relation because:
  we assume an admin can inspect and delete only its products. As a design choice we have decided to implement bidirectionality even if we are not using it, to have a symmetrical implementation.

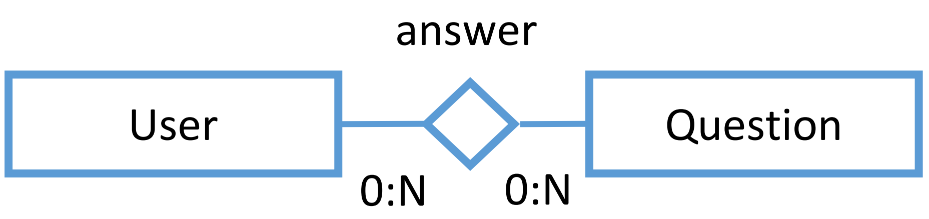
Relationship “reward”

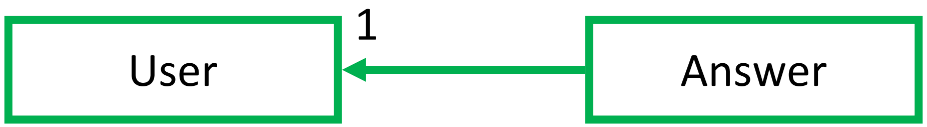


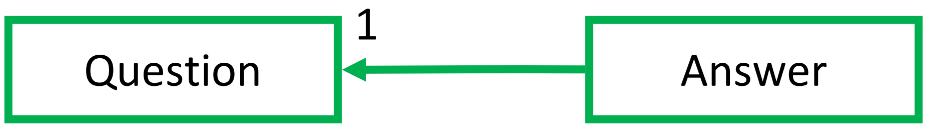
- @ManyToMany
- Modeled as bidirectional relation because:

  From User we need to know if he has already submitted the questionnaire.
- On the opposite side:
- In inspection page we need to access (given a product) the list of users that have completed / deleted the questionnaire *(0 points are assigned if deleted).*

Relationship “answer”



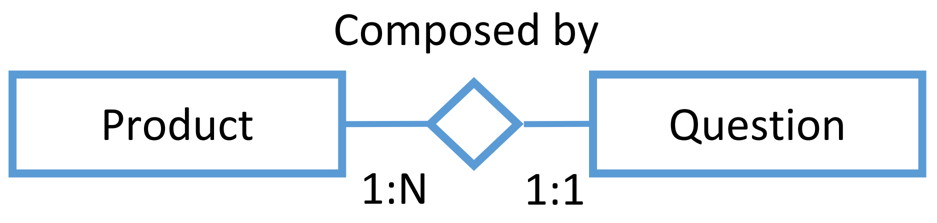


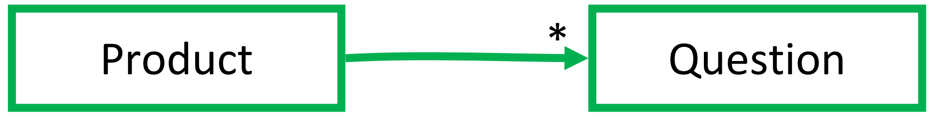


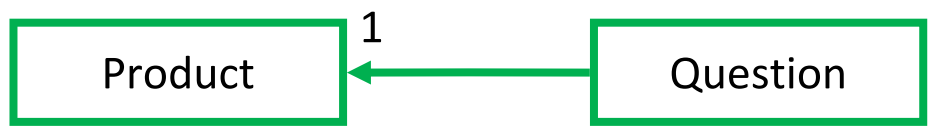
- We modeled the ManyToMany relation between User and Question adding a new entity (i.e. Answer)
- Answer has two unidirectional relations, one with User and one with Question
- Both of them are @ManyToOne
- Because:
  - For each user there are multiple answers
  - For each question there are multiple answers
- Modeled as unidirectional relations because:

  Given an answer:
  - We could need to fetch his "author".
  - We could need to fetch the related question.

Relationship “composed by”



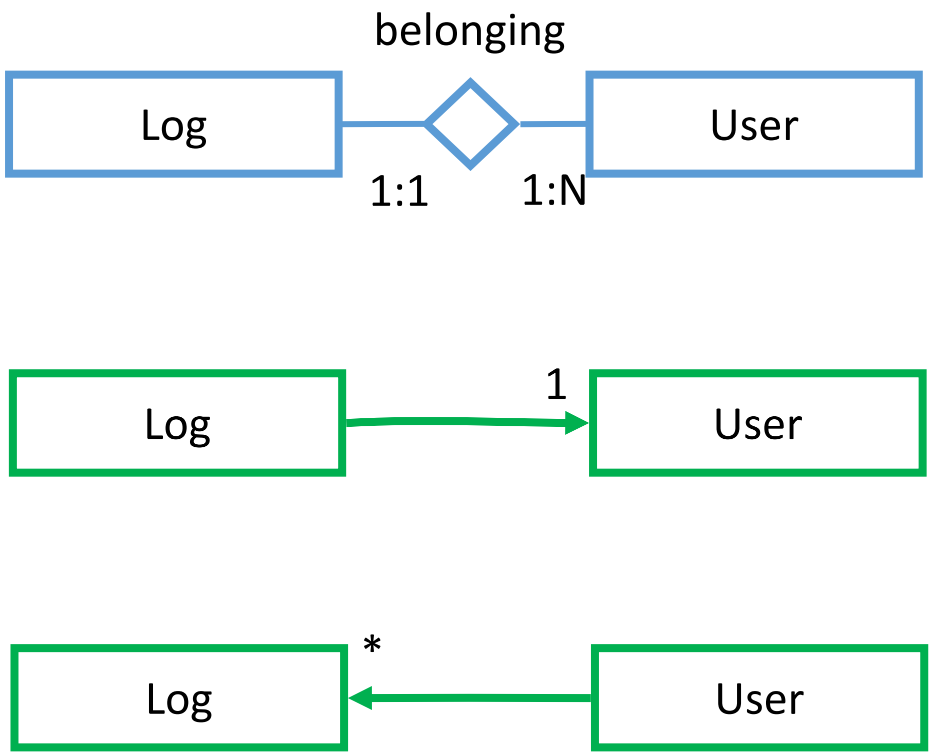




- On Product: @OneToMany
- Because there are several questions related to a product
- On Question: @ManyToOne
- A question is identified by a specific product (weak entity)

- Modeled as bidirectional relation because:
    - Given a product we need to retrieve a list of questions / question.
    - GetProduct() is used in a business method, so we need also to retrieve a product given the question referring to it.

Relationship “belonging”

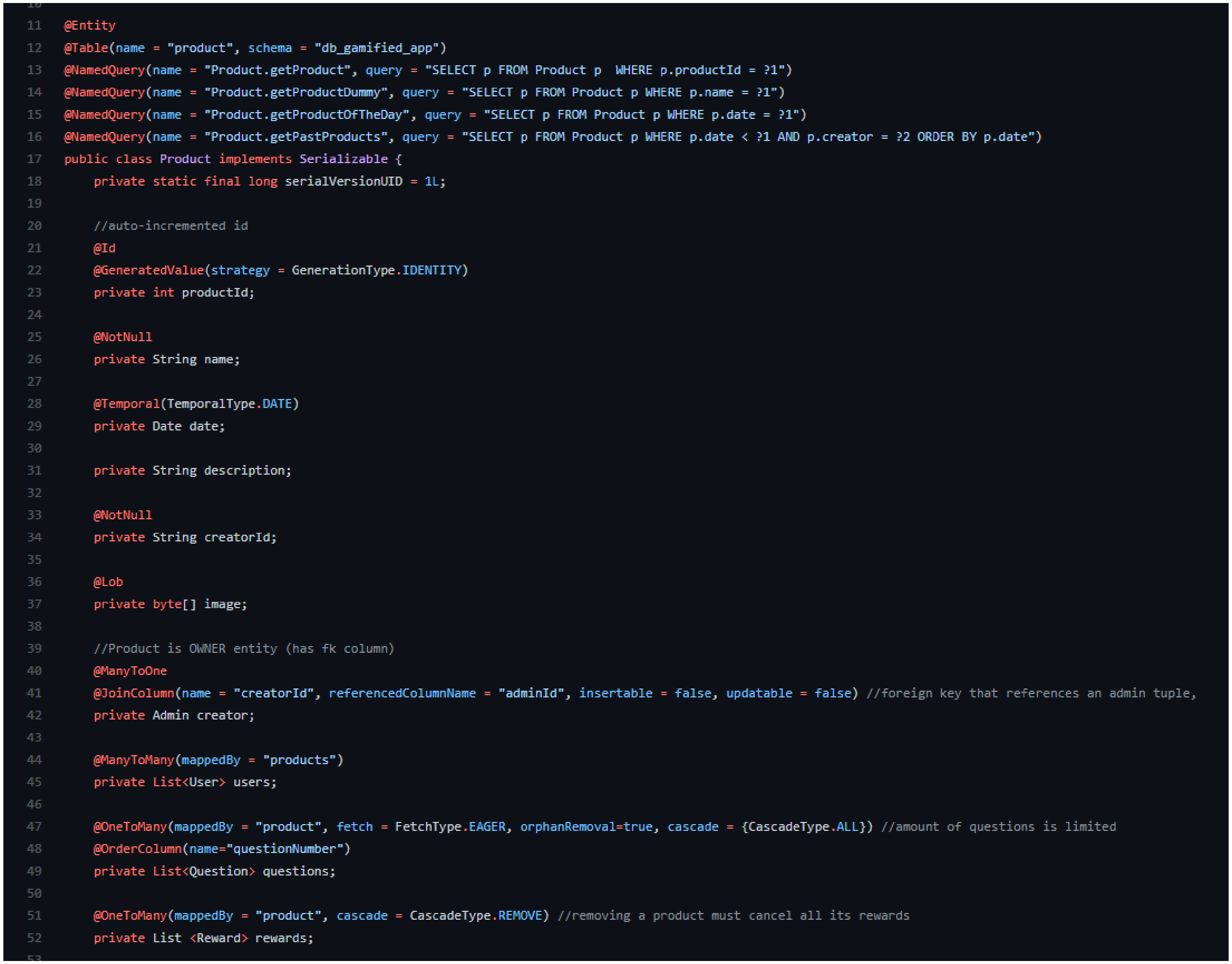


- On Log: @ManyToOne
- Many logs that refer to one user.
- On User: @OneToMany
- Each time a user logs in, a log is saved in the database.
- So a user is associated with several logs. A log is not shared between users.
- We never access this table in the application, we keep it on the database for manual access and checks.
  So only creation operation is exposed.

Entities

- In the coming slides we better describe the main entities of the application. Note that Reward and Answer are modeled as entities too. We do not show DirtyWord, Log and Review entities since they are trivial and less relevant.
- When we do not specify the fetch policy on collection-valued relationships, it means that the default lazy policy is the most appropriate in that case.
- When we do not specify the cascading policy, it means that the cascading is optional, so it can also be left to the default (no cascading)

Entity Product



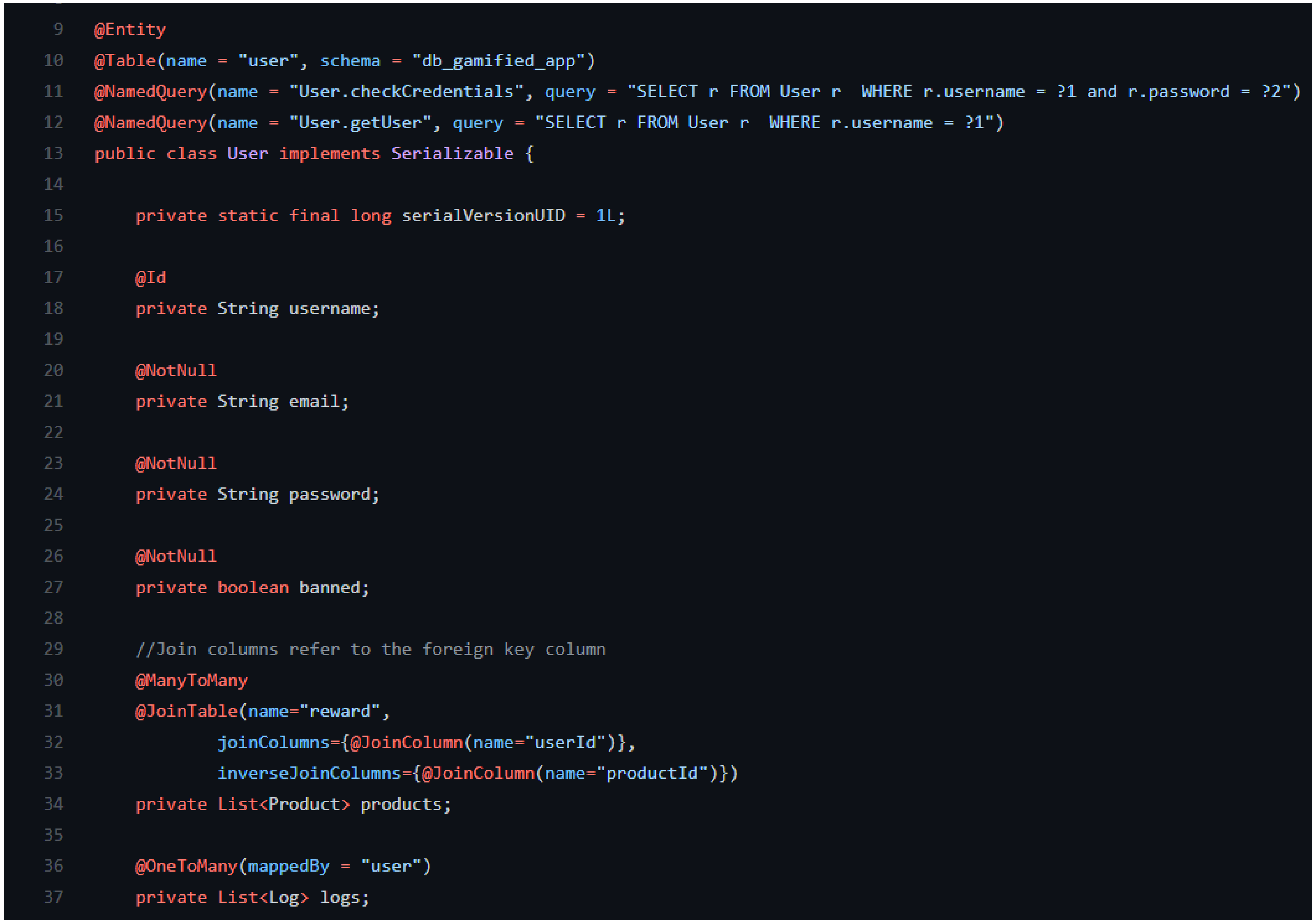
```java
@Entity
@Table(name = "product", schema = "db_gamified_app")
@NamedQuery(name = "Product.getProduct", query = "SELECT p FROM Product p  WHERE p.productId = ?1")
@NamedQuery(name = "Product.getProductDummy", query = "SELECT p FROM Product p WHERE p.name = ?1")
@NamedQuery(name = "Product.getProductOfTheDay", query = "SELECT p FROM Product p WHERE p.date = ?1")
@NamedQuery(name = "Product.getPastProducts", query = "SELECT p FROM Product p WHERE p.date < ?1 AND p.creator = ?2 ORDER BY p.date")
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;

    @Temporal(TemporalType.DATE)
    private Date date;

    private String description;

    @NotNull
    private String creatorId;

    @Lob
    private byte[] image;

    //Product is OWNER entity (has fk column)
    @ManyToOne
    @JoinColumn(name = "creatorId", referencedColumnName = "adminId", insertable = false, updatable = false) //foreign key that references an admin tuple,
    private Admin creator;

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

    @OneToMany(mappedBy = "product", fetch = FetchType.EAGER, orphanRemoval=true, cascade = {CascadeType.ALL}) //amount of questions is limited
    @OrderColumn(name="questionNumber")
    private List<Question> questions;

    @OneToMany(mappedBy = "product", cascade = CascadeType.REMOVE) //removing a product must cancel all its rewards
    private List <Reward> rewards;
```

Motivations

- Bidirectional one-to-many association Question to Product
  - Owner class is Question
  - **CascadeType.ALL** is cascaded because Question is a weak entity of Product, so every change to Product needs to be cascaded to Question
  - **Orphan removal** is set to true because a Question cannot exist without the relative Product, so if we delete a product we also delete all the linked questions and answers
  - Fetch type is **EAGER** because, if the user logs in, he would probably access to the questionnaire of the day, so when we fetch the product, we fetch also the questions right away. The relationship can be navigated at the client side immediately after checking/loading the product entity.
  - **OrderColumn** on questionNumber. We have a specific column in the Question table to be able to order the questions as the admin has inserted them.

- Bidirectional one-to-many association Reward to Product
  - Owner class is Reward
  - **CascadeType.DELETE** is cascaded because when we remove a product, we also remove all the associated rewards.
- Bidirectional many-to-one association Admin to Product
  - Owner class is Product
  - **insertable=false, updatable=false** because it's not the responsibility of the Product entity to create or update an Admin, it is the other way round.

Entity User

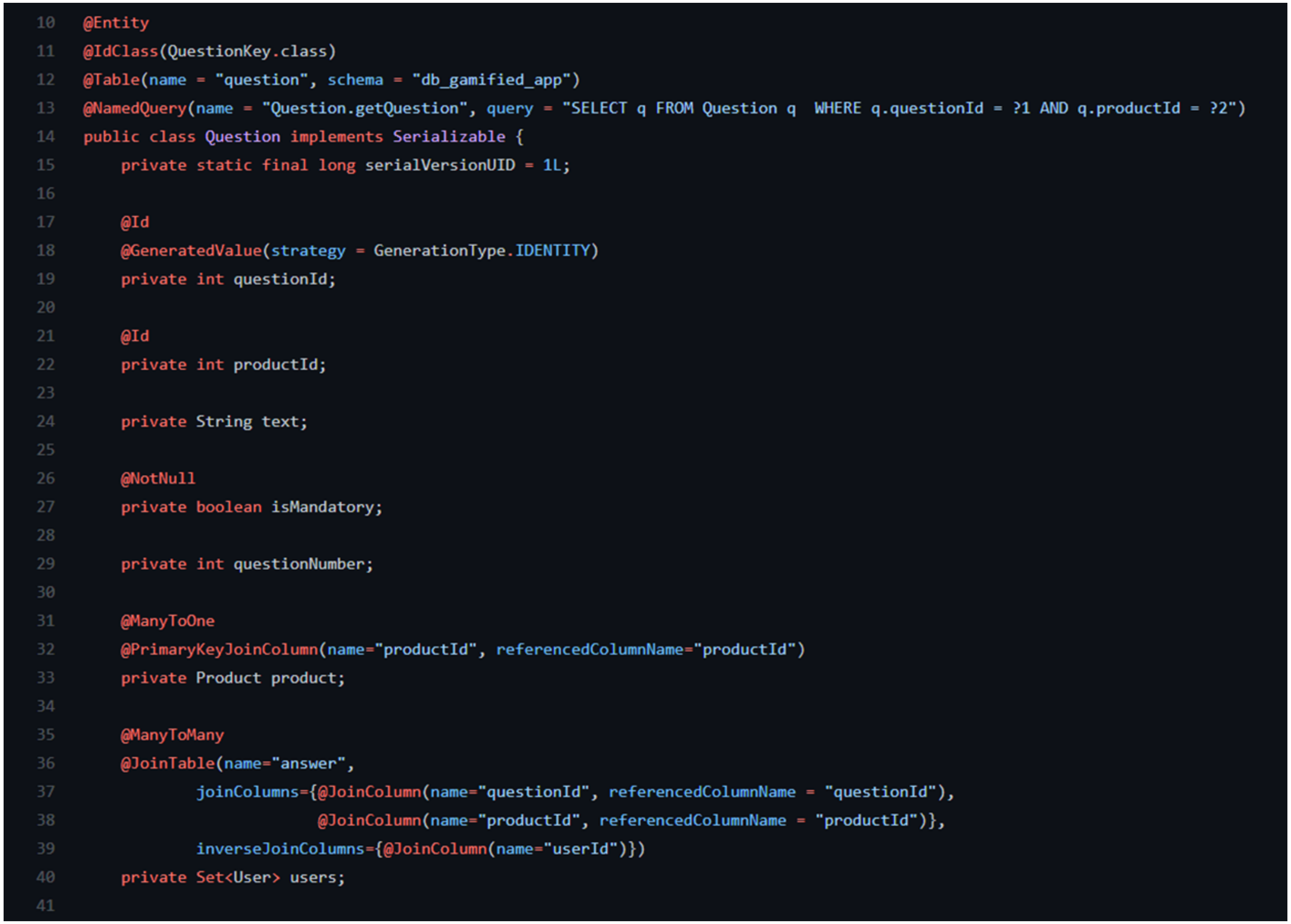


```
@Entity
@Table(name = "user", schema = "db_gamified_app")
@NamedQuery(name = "User.checkCredentials", query = "SELECT r FROM User r  WHERE r.username = ?1 and r.password = ?2")
@NamedQuery(name = "User.getUser", query = "SELECT r FROM User r  WHERE r.username = ?1")
public class User implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String username;

    @NotNull
    private String email;

    @NotNull
    private String password;

    @NotNull
    private boolean banned;

    //Join columns refer to the foreign key column
    @ManyToMany
    @JoinTable(name="reward",
            joinColumns={@JoinColumn(name="userId")},
            inverseJoinColumns={@JoinColumn(name="productId")})
    private List<Product> products;

    @OneToMany(mappedBy = "user")
    private List<Log> logs;
```

Motivations

- Bidirectional many-to-many association Product to User
  - Owner class is User
  - JoinTable is Reward that has as primary keys userId and productId
- Bidirectional one-to-many association Log to User
  - Owner class is Log
  - Operation cascading is optional, indeed is left to the default (no cascading)

Entity Question

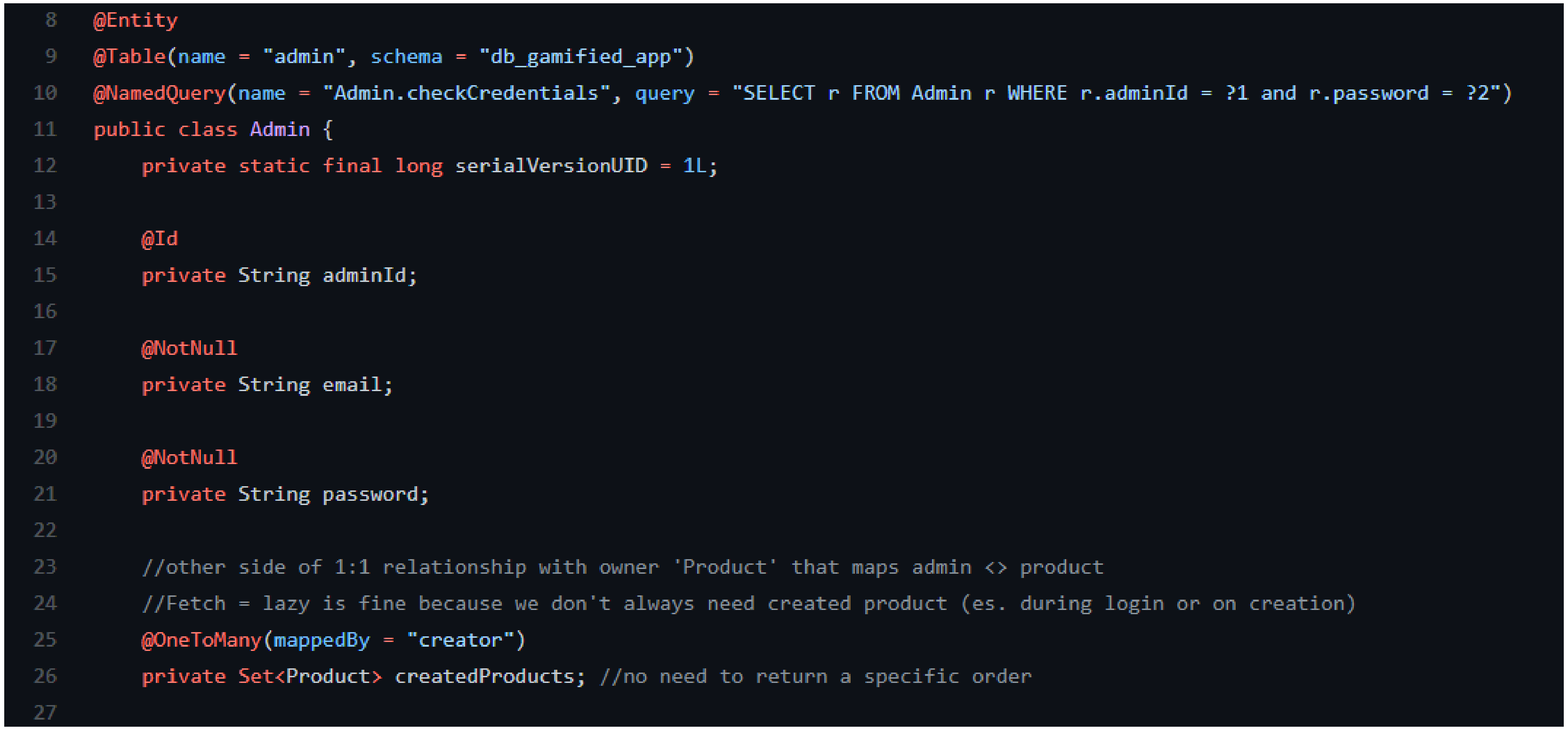


```java
@Entity
@IdClass(QuestionKey.class)
@Table(name = "question", schema = "db_gamified_app")
@NamedQuery(name = "Question.getQuestion", query = "SELECT q FROM Question q  WHERE q.questionId = ?1 AND q.productId = ?2")
public class Question implements Serializable {
    private static final long serialVersionUID = 1L;

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

    @Id
    private int productId;

    private String text;

    @NotNull
    private boolean isMandatory;

    private int questionNumber;

    @ManyToOne
    @PrimaryKeyJoinColumn(name="productId", referencedColumnName="productId")
    private Product product;

    @ManyToMany
    @JoinTable(name="answer",
            joinColumns={@JoinColumn(name="questionId", referencedColumnName = "questionId"),
                         @JoinColumn(name="productId", referencedColumnName = "productId")},
            inverseJoinColumns={@JoinColumn(name="userId")})
    private Set<User> users;

```

Motivations

- Bidirectional many-to-one association Product to Question
  - Owner class is Question
  - **@PrimaryKeyJoinColumn** we are using this annotation to specifiy a primary key column that is used as a foreign key.
- Unidirectional many-to-many association User to Question
  - Owner class is Question
  - JoinTable is Answer that has as primary keys questionId, productid and userId

Entity Admin

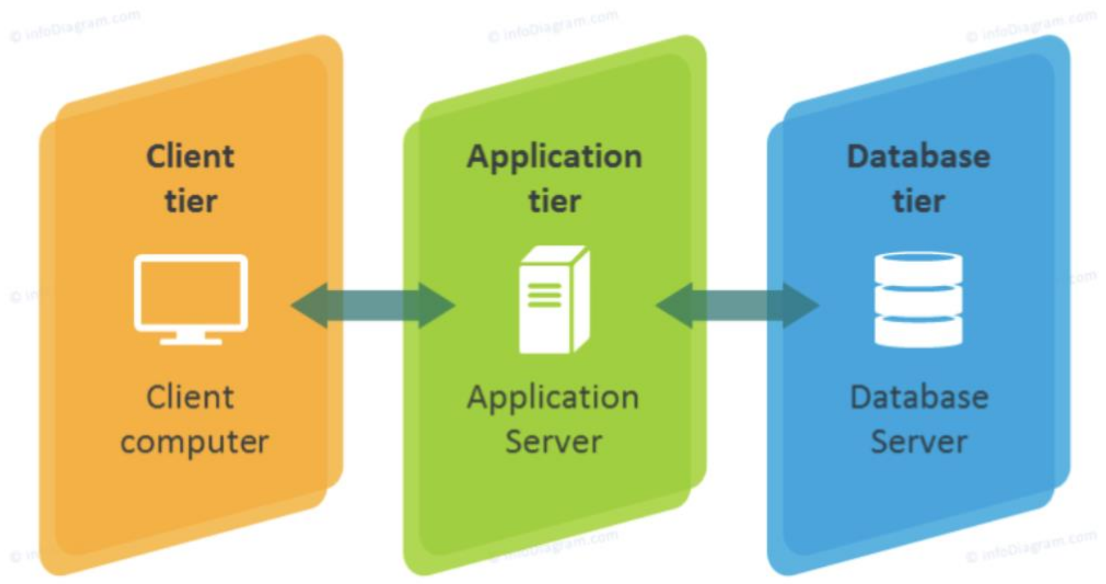


```java
@Entity
@Table(name = "admin", schema = "db_gamified_app")
@NamedQuery(name = "Admin.checkCredentials", query = "SELECT r FROM Admin r WHERE r.adminId = ?1 and r.password = ?2")
public class Admin {
    private static final long serialVersionUID = 1L;

    @Id
    private String adminId;

    @NotNull
    private String email;

    @NotNull
    private String password;

    //other side of 1:1 relationship with owner 'Product' that maps admin <> product
    //Fetch = lazy is fine because we don't always need created product (es. during login or on creation)
    @OneToMany(mappedBy = "creator")
    private Set<Product> createdProducts; //no need to return a specific order

```

Entities Reward and Answer

- Reward and Answer are two relationships with attributes in the ER-diagram, so we modeled them as entities with a primary composite key using the **@EmbeddedId** annotation.
- They have unidirectional many-to-one relationships mapped through the **@mapsId** annotation. Reward has a relation with Product and User, Answer has a relation with Question and User.

Components



Web Tier: **Servlets**

Business Tier: **EJBs**

Business tier components -1

- **@Stateless UserService**
  - `public User insertUser(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, ProductService productService)`
  - `public void LogUser(User user)`

Business tier components -2

- **@Stateless ProductService**
  - `public Product insertProduct(String name, Date date, String description, Admin admin)`
  - `public Product getProduct(int productId)`
  - `public Product getProductOfTheDay()`
  - `public List<Product> getPastQuestionnaires(Date currentDate, Admin creator)`
  - `public List<Answer> getUserAnswers(Product product, String user)`
  - `public List<User> getProductUsers(Product product, Boolean QuestFilled)`
  - `public static List<String> getQuestions(Product product)`
  - `public void dummyImageLoad(int productId, byte[] img)`
  - `public void addCancelledUser(Product product, User user)`
  - `public void deleteProduct(Product product)`
  - `public List<String> getOrderedAnswers(List<Question> questions, List<Answer> answers)`

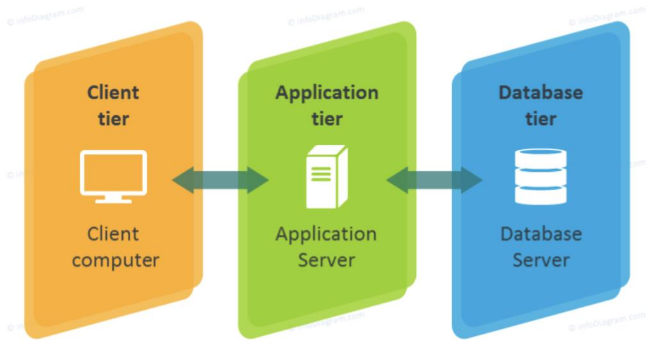
Business tier components -3

- **@Stateless AdminService**
  - `public Admin getAdmin(String adminId)`
  - `public Admin checkAdminCredentials(String adminId, String password)`
  - `public boolean checkQuestionnaireValidity(Date date)`
- **@Stateless QuestionService**
  - `public List<Question> addStatQuestions(List<Question> questions)`
  - `public List<Question> getAllQuestions(List<String> mandatory, Product product)`
  - `public void updateProductQuestions(Product product, List<Question> questions)`
- **@Stateless RewardService**
  - `public List<Reward> getLeaderboard(Product product)`
- **@Stateless AnswerService**
  - `public boolean multipleWordsCheck(String string)`
  - `public void insertAnswer(User user, Question question, String text)`
- **@Stateless ReviewService**
  - `public ArrayList<String> getRandomReviews()`

Motivations

- The business components are stateless because all client requests are served independently and update the database. No main memory conversational state needs to be maintained.

Client components



- **Login / Logout**
- **Error** => get error page data through a json file

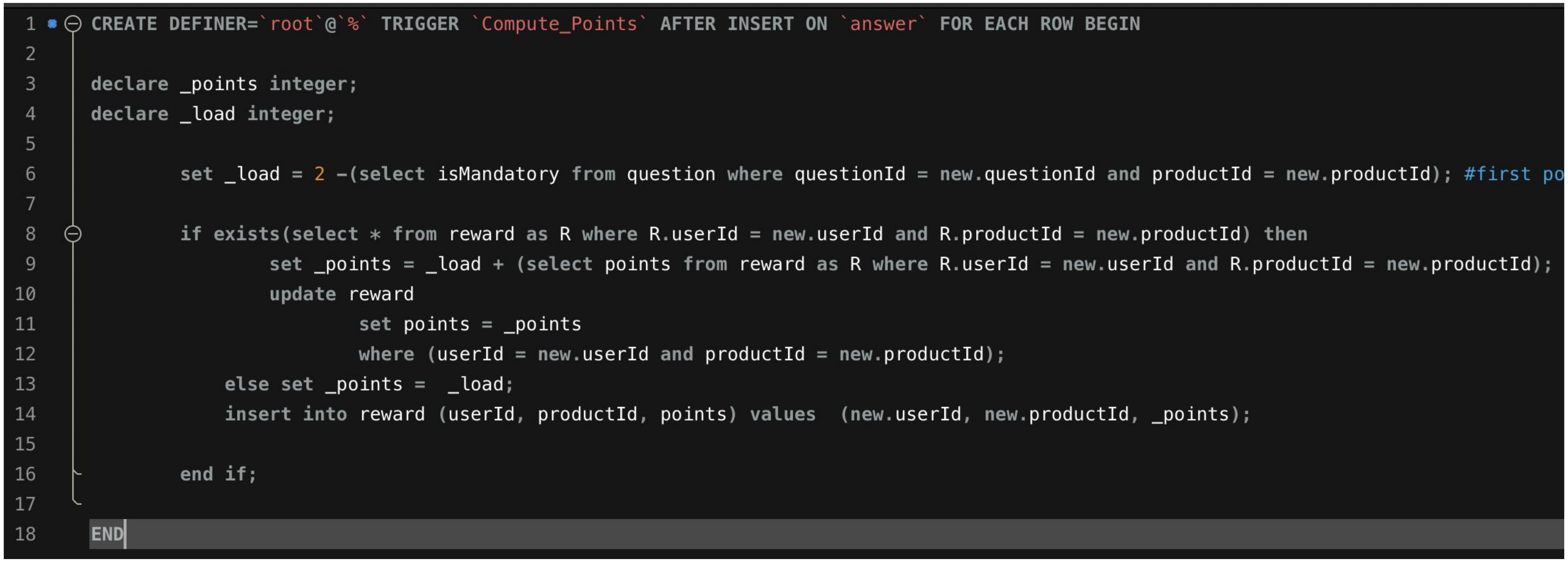
- User side
  - **Homepage** => get user homepage data through a json file
  - **Leaderboard** => get leaderboard page data through a json file
  - **Questionnaire** => get questionnaire page data through a json file
  - **SubmitAnswer** => post user's answers
- Admin side
  - **AdminHomepage** => get admin homepage data through a json file
  - **PastQuestionnaires** => get past questionnaires page data through a json file
  - **Inspection** => get inspection page data through a json file
  - **CreateQuestionnaire** => post newly created product
  - **DeleteQuestionnaire** => post deleted product
  - **UploadImage** => post image

Front-End

- The application front-end has been developed using HTML, that handles the static part and the structure of the pages. Moreover, to handle the dynamic parts we use **JavaScript**. JS employes AJAX calls to request 'page content' and uses the json response to fill the pages html, for example:
  - current username or email, product information…
  - fill an empty table or create one dynamically using iteration on the received json

Gamification Trigger

- For each inserted answer it computes the points. Insert a new entry on reward if it does not exist, update value if does.
  We are using an after trigger because we need to write additional tables as an effect of an insert on a single table, as a form of Active Database.



```
CREATE DEFINER=`root`@`%` TRIGGER `Compute_Points` AFTER INSERT ON `answer` FOR EACH ROW BEGIN

declare _points integer;
declare _load integer;

        set _load = 2 -(select isMandatory from question where questionId = new.questionId and productId = new.productId); #first po

        if exists(select * from reward as R where R.userId = new.userId and R.productId = new.productId) then
                set _points = _load + (select points from reward as R where R.userId = new.userId and R.productId = new.productId);
                update reward
                        set points = _points
                        where (userId = new.userId and productId = new.productId);
            else set _points =  _load;
            insert into reward (userId, productId, points) values  (new.userId, new.productId, _points);

        end if;

END
```

Team Members

Alessandro Lisi – 10621058
alessandro.lisi@mail.polimi.it

Andrea Menta – 10636205
andrea.menta@mail.polimi.it

Lorenzo Mainetti – 10622242
lorenzo.mainetti@mail.polimi.it