Database Specification Component

Last edited by **Tatiana Wang Lin** 5 months ago

EBD: Database Specification Component

askFEUP aims to create a collaborative online platform where FEUP students can ask questions, share experiences, and connect on academic and social topics. Our goal is to build a user-friendly, mobile-responsive space where students can seek advice, offer help, and engage with peers in a supportive community.

The platform will feature categorized questions, personalized content via a "For You" page, and interactive tools like upvoting and leaderboards to encourage participation. With clear roles for administrators, moderators, and users, askFEUP will foster a respectful, well-moderated environment, becoming a central hub for student engagement and collaboration.

A4: Conceptual Data Model

This section provides the identification and description of the entities and relationships within the askFEUP project and its database specification.

1. Class diagram

Figure 5 presents the UML class diagram with the main organizational entities, the relationships between them, attributes and their domains, and the multiplicity of relationships for the askFEUP platform.

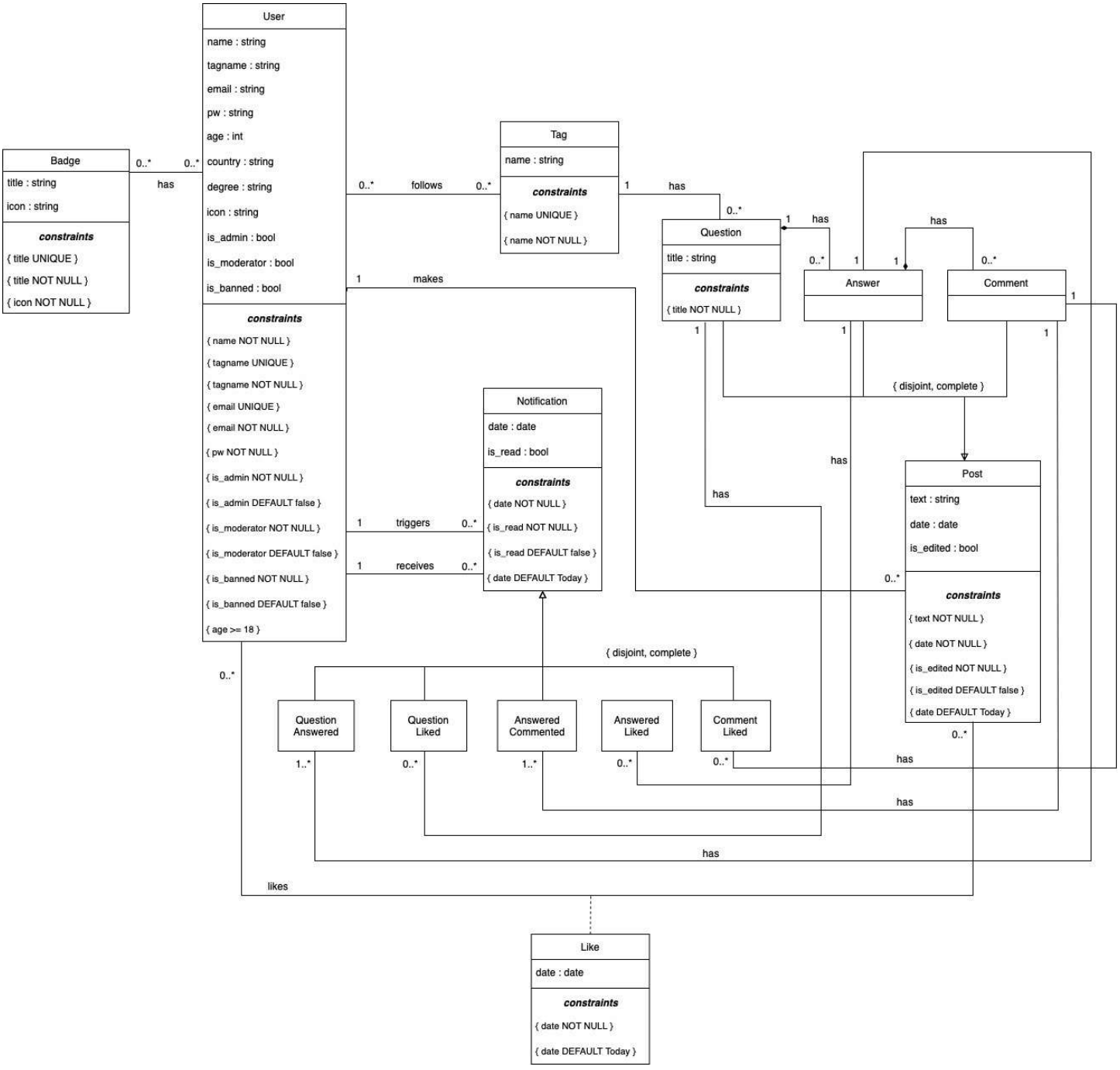

Figure 5: askFEUP Conceptual Data Model in UML

2. Additional Business Rules

Table 13 presents additional business rules not conveyed in the UML class diagram of askFEUP system.

Identifier	Name	Description
BR08	Self-Liking	An user cannot like their own post
BR09	Single Like	An user can only like one time a post
BR10	Chronological Integrity of Post Elements	The date of each question must precede any answers, and each answer's date must precede any related comments
BR11	Self-Reporting Restriction	Users are not permitted to submit reports about themselves

Table 13: Additional Business Rules

A5: Relational Schema, validation and schema refinement

This section presents the Relational Schema derived from the Conceptual Data Model for the askFEUP platform.

1. Relational Schema

Relation reference	Relation Compact Notation
R01	user(id, name **NN**, tagname **UK NN**, email **NN**, pw **NN**, age **CK** age >= 18, country, degree, icon, is_admin **NN DF** False, is_moderator **NN DF** False, is_banned **NN DF** False)
R02	badge(id, title **UK NN**, icon **NN**)
R03	user_badge(user_id → user **NN**, badge_id → badge **NN**)
R04	tag(id, name **UK NN**)
RO5	question_post(id, title **NN**, text **NN**, date **NN DF** Today, is_edited **NN DF** False, nr_likes **NN DF** 0 **CK** nr_likes >= 0, user_id → user **NN**, tag_id → tag **NN**)
RO6	answer_post(id, text **NN**, date **NN DF** Today, is_edited **NN DF** False, nr_likes **NN DF** 0 **CK** nr_likes >= 0, user_id → user **NN**, question_id → question_post **NN**)
RO7	comment_post(id, text **NN**, date **NN DF** Today, is_edited **NN DF** False, nr_likes **NN DF** 0 **CK** nr_likes >= 0, user_id → user **NN**, answer_id → answer_post **NN**)
R08	question_like(user_id → user **NN**, post_id -> question_post **NN**, date **NN DF** Today)
R09	answer_like(user_id → user **NN**, post_id -> answer_post **NN**, date **NN DF** Today)
R10	comment_like(user_id → user **NN**, post_id -> comment_post **NN**, date **NN DF** Today)
R11	question_answered_notification(id, user_receive_id → user **NN**, date **NN**, is_read **NN**, user_trigger_id → user **NN**, question_id → question_post **NN**, answer_id → answer_post **NN**)
R12	question_liked_notification(id, user_receive_id → user **NN**, date **NN**, is_read **NN**, user_trigger_id → user **NN**, question_id → question_post **NN**)
R13	answer_commented_notification(id, user_receive_id → user **NN**, date **NN DF** Today, is_read **NN**, user_trigger_id → user **NN**, answer_id → answer **NN**, comment_id → comment_post **NN**)
R14	answer_liked_notification(id, user_receive_id → user **NN**, date **NN DF** Today <= Today, is_read **NN**, user_trigger_id → user **NN**, answer_id → answer_post **NN**)
R15	comment_liked_notification(id, user_receive_id → user **NN**, date **NN DF** Today, is_read **NN DF** False, user_trigger_id → user **NN**, comment_id → comment_post **NN**)
R16	user_follow_tag(user_id → user **NN**, tag_id → tag **NN**)

Table 14: askFEUP Relational Schema

Legend:

- **UK** → UNIQUE

- **NN** → NOT NULL
- **DF** → DEFAULT
- **CK** → CHECK

2. Domains

Table 15 represents the domains used in our UML representation.

Domain Name	Domain Specification
Today	DATE DEFAULT CURRENT_DATE

Table 15: askFEUP Domains

3. Schema validation

To validate the Relational Schema obtained from the Conceptual Data Model, all functional dependencies are identified and the normalization of all relation schemas is accomplished.

TABLE R01	user
Keys	{ id }, { tagname }, { email }
Functional Dependencies:	
FD0101	id → { name, tagname, email, pw, age, country, degree, icon, is_admin, is_moderator, is_banned }
FD0102	tagname → { id, name, email, pw, age, country, degree, icon, is_admin, is_moderator, is_banned }
FD0103	email → { id, name, tagname, pw, age, country, degree, icon, is_admin, is_moderator, is_banned }
NORMAL FORM	BCNF

Table 16: user schema validation

TABLE R02	badge
Keys	{ id, title }
Functional Dependencies:	
FD0201	id → { title, icon }
FD0201	title → { id, icon }
NORMAL FORM	BCNF

Table 17: badge schema validation

TABLE R03	user_badge
Keys	{ user_id, badge_id }
Functional Dependencies:	None
NORMAL FORM	BCNF

Table 18: user_badge schema validation

TABLE R04	tag
Keys	{ id, name }
Functional Dependencies:	
FD0401	id → { name }
FD0402	name → { id }
NORMAL FORM	BCNF

Table 19: tag schema validation

TABLE R05	question_post
Keys	{ id }
Functional Dependencies:	
FD0501	id → { title, text, date, is_edited, nr_likes, user_id, tag_id }
NORMAL FORM	BCNF

Table 20: question_post schema validation

TABLE R06	answer_post
Keys	{ id }
Functional Dependencies:	
FD0601	id → { text, date, is_edited, nr_likes, user_id, question_id }
NORMAL FORM	BCNF

Table 21: answer_post schema validation

TABLE R07	comment_post
Keys	{ id }
Functional Dependencies:	
FD0701	id → { text, date, is_edited, nr_likes, user_id, answer_id }
NORMAL FORM	BCNF

Table 22: comment_post schema validation

TABLE R08	question_like
Keys	{ user_id, post_id }
Functional Dependencies:	
FD0801	user_id, post_id → { date }
NORMAL FORM	BCNF

Table 23: question_like schema validation

TABLE R09	answer_like
Keys	{ user_id, post_id }
Functional Dependencies:	
FD0801	user_id, post_id → { date }
NORMAL FORM	BCNF

Table 24: answer_like schema validation

TABLE R10	comment_like
Keys	{ user_id, post_id }
Functional Dependencies:	
FD0801	user_id, post_id → { date }

TABLE R10	comment_like
NORMAL FORM	BCNF

Table 25: comment_like schema validation

TABLE R11	question_answered_notification
Keys	{ id, question_id, answer_id }
Functional Dependencies:	
FD0901	id → { user_receive_id, date, is_read, user_trigger_id, question_id, answer_id }
FD0901	{ question_id, answer_id } → {id, date, is_read, user_receive_id, user_trigger_id }
NORMAL FORM	BCNF

Table 26: question_answered_notification schema validation

TABLE R12	question_liked_notification
Keys	{ id, question_id, user_trigger_id }
Functional Dependencies:	
FD1001	id → { user_receive_id, date, is_read, user_trigger_id, question_id }
FD1001	{ question_id, user_trigger_id } → { id, date, is_read, user_receive_id }
NORMAL FORM	BCNF

Table 27: question_liked_notification schema validation

TABLE R13	answer_commented_notification
Keys	{ id, answer_id, comment_id }
Functional Dependencies:	
FD1101	id → { user_receive_id, date, is_read, user_trigger_id, answer_id, comment_id }
FD1101	{ answer_id, comment_id } → { id, date, is_read, user_receive_id, user_trigger_id }
NORMAL FORM	BCNF

Table 28: answer_commented_notification schema validation

TABLE R14	answer_liked_notification
Keys	{ id, answer_id, user_trigger_id }
Functional Dependencies:	
FD1201	id → { user_receive_id, date, is_read, user_trigger_id, answer_id }
FD1201	{ answer_id, user_trigger_id } → {id, date, is_read, user_receive_id }
NORMAL FORM	BCNF

Table 29: answer_liked_notification schema validation

TABLE R15	comment_liked_notification
Keys	{ id, comment_id, user_trigger_id }
Functional Dependencies:	
FD1301	id → { user_receive_id, date, is_read, user_trigger_id, comment_id }

TABLE R15	comment_liked_notification
FD1301	{ comment_id, user_trigger_id } → { id, date, is_read, user_receive_id }
NORMAL FORM	BCNF

Table 30: comment_liked_notification schema validation

TABLE R16	user_follow_tag
Keys	{ user_id, tag_id }
Functional Dependencies:	
FD1401	None
NORMAL FORM	BCNF

Table 31: user_follow_tag schema validation

A6: Indexes, triggers, transactions and database population

This section presents the Indexes, Triggers, Transactions, and Database Population of the askFEUP system.

1. Database Workload

The database workload is presented in Table 32. It provides an estimated tuple count for each relation, along with the estimated growth.

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	user	100k	1000/day
R02	badge	10	No Growth
R03	user_badge	100	10/month
R04	tag	10	1/month
R05	question_post	1k	10/day
R06	answer_post	10k	100/day
R07	comment_post	10k	100/day
R08	question_like	10k	100/day
R09	answer_like	10k	100/day
R10	comment_like	10k	100/day
R11	question_answered_notification	10k	100/day
R12	question_liked_notification	10k	100/day
R13	answered_commented_notification	10k	100/day
R14	answer_liked_notification	10k	100/day
R15	comment_liked_notification	10k	100/day
R16	user_follow_tag	10k	100/day

Table 32: askFEUP database workload

2. Proposed Indices

2.1. Performance Indices

Index	IDX01
Relation	question_post
Attribute	tag_id
Type	Hash
Cardinality	medium
Clustering	no
Justification	The table "question" is really large and many users will want to search questions by their tag. This is a requirement for our project. Filtering is done by exact match, making an hash type index the best option.
SQL code	`CREATE INDEX idx_question_tag ON question_post USING hash (tag_id)`

Table 33: Index for question_post table for the tag id

Index	IDX02
Relation	question_post
Attribute	nr_likes
Type	BTree
Cardinality	low
Clustering	yes
Justification	We want to present to the user question posts that are really popular, for a better experience. Therefore, it's important to create an index using BTree, as the number of likes is countable and searching for the most popular posts will be more efficient.
SQL code	`CREATE INDEX idx_question_likes ON question_post (nr_likes)`

Table 34: Index for question_post table for the number of likes

Index	IDX03
Relation	question_post
Attribute	post_date
Type	BTree
Cardinality	medium
Clustering	no
Justification	We want to present the posts sorted by date. Usually, we want to present the most recent posts to lure more answers, creating a better experience.
SQL code	`CREATE INDEX idx_question_date ON question_post (date)`

Table 35: Index for question_post table for the date of the post

2.2. Full-text Search Indices

Index	IDX04
Relation	question_post
Attribute	tsvectors
Type	GIN
Clustering	no

Index	IDX04
Justification	We want to give the user the opportunity to search any question by writing a text as input. GIN is the better option here, since the title of the questions will not change frequently.
SQL code	`CREATE INDEX idx_question_title_search ON question_post USING GIN (tsvectors)`

Table 36: Index for the question_post table for the FTS

3. Triggers

Trigger	TRIGGER01
Description	Update nr_likes on post when a new like is added.

SQL Code

```
CREATE OR REPLACE FUNCTION update_post_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE question_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_like_trigger
AFTER INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION update_post_like();
```

Trigger	TRIGGER02
Description	Prevent user from liking their own post. This is addressed by the business rule BR.105.

SQL Code

```
CREATE OR REPLACE FUNCTION prevent_self_like() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM question_post WHERE id = NEW.post_id) = NEW.user_id THEN
        RAISE EXCEPTION 'Cannot like own post';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_self_like_trigger
BEFORE INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like();
```

Trigger	TRIGGER03
Description	Update nr_likes on post when a like is removed.

SQL CODE

```
CREATE OR REPLACE FUNCTION update_post_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE question_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_unlike_trigger
AFTER DELETE ON question_like
```

```
FOR EACH ROW
EXECUTE FUNCTION update_post_unlike();
```

Trigger	TRIGGER04
Description	Update nr_likes on answer post when a new like is added.

SQL Code

```
CREATE OR REPLACE FUNCTION update_answer_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE answer_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_like_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION update_answer_like();
```

Trigger	TRIGGER05
Description	Prevent user from liking their own answer. This is addressed by the business rule BR.105.

SQL Code

```
CREATE OR REPLACE FUNCTION prevent_self_like_answer() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM answer_post WHERE id = NEW.post_id) = NEW.user_id THEN
        RAISE EXCEPTION 'Cannot like own answer';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_self_like_trigger
BEFORE INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like_answer();
```

Trigger	TRIGGER06
Description	Update nr_likes on answer post when a like is removed.

SQL Code

```
CREATE OR REPLACE FUNCTION update_answer_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE answer_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_unlike_trigger
AFTER DELETE ON answer_like
FOR EACH ROW
EXECUTE FUNCTION update_answer_unlike();
```

Trigger	TRIGGER07
Description	Update nr_likes on comment post when a new like is added

SQL Code

```
CREATE OR REPLACE FUNCTION update_comment_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE comment_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_like_trigger
AFTER INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION update_comment_like();
```

Trigger	TRIGGER08
Description	Prevent user from liking their own comment. This is addressed by the business rule BR.105.

SQL Code

```
CREATE OR REPLACE FUNCTION prevent_self_like_comment() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM comment_post WHERE id = NEW.post_id) = NEW.user_id THEN
        RAISE EXCEPTION 'Cannot like own comment';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_self_like_trigger
BEFORE INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like_comment();
```

Trigger	TRIGGER09
Description	Update nr_likes on comment post when a like is removed.

SQL Code

```
CREATE OR REPLACE FUNCTION update_comment_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE comment_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_unlike_trigger
AFTER DELETE ON comment_like
FOR EACH ROW
EXECUTE FUNCTION update_comment_unlike();
```

Trigger	TRIGGER10
Description	Create a notification when a question is answered.

SQL Code

```
CREATE OR REPLACE FUNCTION notify_question_answered() RETURNS TRIGGER AS $$
DECLARE
    question_author INT;
BEGIN
    SELECT user_id INTO question_author
    FROM question_post
```

```
    WHERE id = NEW.question_id;

    IF question_author IS NOT NULL AND EXISTS (SELECT 1 FROM "user" WHERE id = NEW.user_id) THEN
        INSERT INTO question_answered_notification(user_receive_id, user_trigger_id, question_id, answer_id, date)
        VALUES (
            question_author,
            NEW.user_id,
            NEW.question_id,
            NEW.id,
            NOW()
        );
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_answered_notification_trigger
AFTER INSERT ON answer_post
FOR EACH ROW
EXECUTE FUNCTION notify_question_answered();
```

Trigger	TRIGGER11
Description	Create a notification when a question is liked.

SQL Code

```
CREATE OR REPLACE FUNCTION notify_question_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO question_liked_notification(user_receive_id, user_trigger_id, question_id)
    VALUES (
        (SELECT user_id FROM question_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_liked_notification_trigger
AFTER INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION notify_question_liked();
```

Trigger	TRIGGER12
Description	Create a notification when an answer is liked

SQL Code

```
CREATE OR REPLACE FUNCTION notify_answer_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO answer_liked_notification(user_receive_id, user_trigger_id, answer_id)
    VALUES (
        (SELECT user_id FROM answer_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_liked_notification_trigger
```

```
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION notify_answer_liked();
```

Trigger	TRIGGER13
Description	Create a notification when a comment is liked.

SQL Code

```
CREATE OR REPLACE FUNCTION notify_comment_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO comment_liked_notification(user_receive_id, user_trigger_id, comment_id)
    VALUES (
        (SELECT user_id FROM comment_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_liked_notification_trigger
AFTER INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION notify_comment_liked();
```

Trigger	TRIGGER14
Description	Create a notification when a comment is made on an answer.

SQL Code

```
CREATE OR REPLACE FUNCTION notify_answer_commented() RETURNS TRIGGER AS $$
DECLARE
    answer_author INT;
BEGIN
    SELECT user_id INTO answer_author
    FROM answer_post
    WHERE id = NEW.answer_id;

    IF answer_author IS NOT NULL AND EXISTS (SELECT 1 FROM "user" WHERE id = NEW.user_id) THEN
        INSERT INTO answer_commented_notification(user_receive_id, user_trigger_id, answer_id, comment_id, date)
        VALUES (
            answer_author,
            NEW.user_id,
            NEW.answer_id,
            NEW.id,
            NOW()
        );
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_commented_notification_trigger
AFTER INSERT ON comment_post
FOR EACH ROW
EXECUTE FUNCTION notify_answer_commented();
```

Trigger	TRIGGER15
Description	Set an user as null but keep their interactions

SQL Code

```
CREATE OR REPLACE FUNCTION anonymize_user()
RETURNS TRIGGER AS $$
BEGIN
    -- Update user data to anonymized values instead of deleting
    UPDATE "user"
    SET
        name = 'Anonymous',
        tagname = 'Anonymous_' || OLD.id,
        email = NULL,
        pw = NULL,
        age = NULL,
        country = NULL,
        degree = NULL,
        icon = NULL,
        is_admin = FALSE,
        is_moderator = FALSE,
        is_banned = TRUE
    WHERE id = OLD.id;

    RETURN NULL;  -- Prevent actual DELETE operation
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_anonymize_user
BEFORE DELETE ON "user"
FOR EACH ROW
EXECUTE FUNCTION anonymize_user();
```

Trigger	TRIGGER16
Description	Trigger for the "Expert" Badge (100 Likes Across All Answers)

SQL Code

```
CREATE OR REPLACE FUNCTION assign_badge(p_user_id INT, p_badge_name TEXT) RETURNS VOID AS $$
DECLARE
    v_badge_id INT;
BEGIN
    -- Find the badge ID based on the badge name
    SELECT id INTO v_badge_id
    FROM badge
    WHERE title = p_badge_name;

    -- Insert the badge only if it doesn't already exist for the user
    IF NOT EXISTS (
        SELECT 1 FROM user_badge
        WHERE user_badge.user_id = p_user_id AND user_badge.badge_id = v_badge_id
    ) THEN
        INSERT INTO user_badge (user_id, badge_id) VALUES (p_user_id, v_badge_id);
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION check_expert_badge() RETURNS TRIGGER AS $$
DECLARE
    total_likes INT;
BEGIN
    -- Count total likes across all answers by the user
    SELECT COUNT(*) INTO total_likes
    FROM answer_like l
    JOIN answer_post a ON l.post_id = a.id
    WHERE a.user_id = NEW.user_id;

    -- Assign the Expert badge if total likes reach 100
```

```
    IF total_likes >= 100 THEN
        PERFORM assign_badge(NEW.user_id, 'Expert');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER expert_badge_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION check_expert_badge();
```

Trigger	TRIGGER17
Description	Trigger for the "Contributor" Badge (First Question)

SQL Code

```
CREATE OR REPLACE FUNCTION check_contributor_badge() RETURNS TRIGGER AS $$
BEGIN
    -- Assign the Contributor badge if the user creates their first question
    PERFORM assign_badge(NEW.user_id, 'Contributor');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER contributor_badge_trigger
AFTER INSERT ON question_post
FOR EACH ROW
EXECUTE FUNCTION check_contributor_badge();
```

Trigger	TRIGGER18
Description	Trigger for the "Top Answerer" Badge (50 Likes on a Single Answer)

SQL Code

```
CREATE OR REPLACE FUNCTION check_top_answerer_badge() RETURNS TRIGGER AS $$
DECLARE
    answer_likes INT;
BEGIN
    -- Count likes for the specific answer
    SELECT COUNT(*) INTO answer_likes
    FROM answer_like
    WHERE post_id = NEW.post_id;

    -- Assign the Top Answerer badge if likes reach 50 on this answer
    IF answer_likes >= 50 THEN
        PERFORM assign_badge(NEW.user_id, 'Top Answerer');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER top_answerer_badge_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION check_top_answerer_badge();
```

Trigger	TRIGGER19
Description	Trigger for the "Rising Star" Badge (10 Answers)

SQL Code

```
CREATE OR REPLACE FUNCTION check_rising_star_badge() RETURNS TRIGGER AS $$
DECLARE
    total_answers INT;
BEGIN
    -- Count total answers by the user
    SELECT COUNT(*) INTO total_answers
    FROM answer_post
    WHERE user_id = NEW.user_id;

    -- Assign the Rising Star badge if answers reach 10
    IF total_answers >= 10 THEN
        PERFORM assign_badge(NEW.user_id, 'Rising Star');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER rising_star_badge_trigger
AFTER INSERT ON answer_post
FOR EACH ROW EXECUTE FUNCTION check_rising_star_badge();
```

Trigger	TRIGGER20
Description	Trigger for the "Community Leader" Badge (50 Posts)

SQL Code

```
CREATE OR REPLACE FUNCTION check_community_leader_badge() RETURNS TRIGGER AS $$
DECLARE
    total_posts INT;
BEGIN
    -- Count total posts by the user (questions + answers)
    SELECT
        (SELECT COUNT(*) FROM question_post WHERE user_id = NEW.user_id) +
        (SELECT COUNT(*) FROM answer_post WHERE user_id = NEW.user_id)
    INTO total_posts;

    -- Assign the Community Leader badge if total posts reach 50
    IF total_posts >= 50 THEN
        PERFORM assign_badge(NEW.user_id, 'Community Leader');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER community_leader_badge_trigger_question
AFTER INSERT ON question_post
FOR EACH ROW
EXECUTE FUNCTION check_community_leader_badge();

CREATE TRIGGER community_leader_badge_trigger_answer
AFTER INSERT ON answer_post
FOR EACH ROW
EXECUTE FUNCTION check_community_leader_badge();
```

4. Transactions

Transaction	TRAN01
Description	Insert an answer only if the referenced question exists.

Transaction	TRAN01
Justification	This transaction maintains data integrity by ensuring that answers are only added if a valid question exists. The Read Committed isolation level guarantees that only committed question records are read, preventing interference from incomplete changes.
Isolation level	READ COMMITTED

SQL Code

```
CREATE OR REPLACE FUNCTION AddAnswerIfQuestionExists(
    question_id INT,
    answer_content TEXT,
    user_id INT
)
RETURNS VOID AS $$
BEGIN
    BEGIN
        IF EXISTS (SELECT 1 FROM question_post WHERE id = question_id) THEN
            INSERT INTO answer_post(text, date, is_edited, user_id, nr_likes)
            VALUES (answer_content, NOW(), FALSE, user_id, 0);
        ELSE
            RAISE EXCEPTION 'Question does not exist';
        END IF;

    EXCEPTION
        WHEN OTHERS THEN
            RAISE EXCEPTION 'An error occurred while inserting the answer';
    END;
END;
$$ LANGUAGE plpgsql;
```

Transaction	TRAN02
Description	Insert comment if both question and answer exist
Justification	This transaction maintains data integrity by ensuring that comments are only added if a valid question and answer exist. The Read Committed isolation level guarantees that only committed question and answer records are read, preventing interference from incomplete changes.
Isolation level	READ COMMITTED

SQL Code

```
CREATE OR REPLACE FUNCTION AddCommentIfQuestionAndAnswerExist(
    question_id INT,
    answer_id INT,
    comment_content TEXT,
    user_id INT
)
RETURNS VOID AS $$
BEGIN
    BEGIN
        IF EXISTS (SELECT 1 FROM question_post WHERE id = question_id)
           AND EXISTS (SELECT 1 FROM answer_post WHERE id = answer_id) THEN
            INSERT INTO comment_post(text, date, is_edited, user_id, nr_likes)
            VALUES (comment_content, NOW(), FALSE, user_id, 0);
        ELSE
            RAISE EXCEPTION 'Question or answer does not exist';
        END IF;

    EXCEPTION
        WHEN OTHERS THEN
            RAISE EXCEPTION 'An error occurred while inserting the comment';
    END;
```

```
END;
$$ LANGUAGE plpgsql;
```

Annex A. SQL Code

This section presents the .sql document used for our database.

A.1. Database schema

```
DROP SCHEMA IF EXISTS lbaw24042 CASCADE;
CREATE SCHEMA IF NOT EXISTS lbaw24042;

SET search_path TO lbaw24042;

-- Drop Tables
DROP TABLE IF EXISTS "user";
DROP TABLE IF EXISTS badge;
DROP TABLE IF EXISTS user_badge;
DROP TABLE IF EXISTS tag;
DROP TABLE IF EXISTS question_post;
DROP TABLE IF EXISTS answer_post;
DROP TABLE IF EXISTS comment_post;
DROP TABLE IF EXISTS question_like;
DROP TABLE IF EXISTS answer_like;
DROP TABLE IF EXISTS comment_like;
DROP TABLE IF EXISTS question_answered_notification;
DROP TABLE IF EXISTS question_liked_notification;
DROP TABLE IF EXISTS answer_commented_notification;
DROP TABLE IF EXISTS answer_liked_notification;
DROP TABLE IF EXISTS comment_liked_notification;
DROP TABLE IF EXISTS user_follow_tag;

-- R01: User Table
CREATE TABLE "user" (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    tagname VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    pw VARCHAR(255) NOT NULL,
    age INT CHECK (age >= 18),
    country VARCHAR(100),
    degree VARCHAR(255),
    icon TEXT,
    is_admin BOOLEAN NOT NULL DEFAULT FALSE,
    is_moderator BOOLEAN NOT NULL DEFAULT FALSE,
    is_banned BOOLEAN NOT NULL DEFAULT FALSE
);

-- R02: Badge Table
CREATE TABLE badge (
    id SERIAL PRIMARY KEY,
    title VARCHAR(255) UNIQUE NOT NULL,
    icon TEXT NOT NULL
);

-- R03: User_Badge Table
CREATE TABLE user_badge (
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE,
    badge_id INT REFERENCES badge(id) ON DELETE CASCADE
);

-- R04: Tag Table
CREATE TABLE tag (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) UNIQUE NOT NULL
);
```

```
-- R05: Question_Post Table
CREATE TABLE question_post (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    date TIMESTAMP NOT NULL CHECK (date <= NOW()),
    is_edited BOOLEAN NOT NULL DEFAULT FALSE,
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    title VARCHAR(255) UNIQUE NOT NULL,
    tag_id INT REFERENCES tag(id) ON DELETE SET NULL,
    nr_likes INT NOT NULL CHECK (nr_likes >= 0) DEFAULT 0
);

-- R06: Answer_Post Table
CREATE TABLE answer_post (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    date TIMESTAMP NOT NULL CHECK (date <= NOW()),
    is_edited BOOLEAN NOT NULL DEFAULT FALSE,
        nr_likes INT NOT NULL CHECK (nr_likes >= 0) DEFAULT 0,
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
        question_id INT REFERENCES question_post(id) ON DELETE CASCADE NOT NULL
);

-- R07: Comment_Post Table
CREATE TABLE comment_post (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    date TIMESTAMP NOT NULL CHECK (date <= NOW()),
    is_edited BOOLEAN NOT NULL DEFAULT FALSE,
        nr_likes INT NOT NULL CHECK (nr_likes >= 0) DEFAULT 0,
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    answer_id INT REFERENCES answer_post(id) ON DELETE CASCADE NOT NULL
);

-- R08: Question Like Table
CREATE TABLE question_like (
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE,
    post_id INT REFERENCES question_post(id) ON DELETE CASCADE,
    date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, post_id)
);

-- R09: Answer Like Table
CREATE TABLE answer_like (
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE,
    post_id INT REFERENCES answer_post(id) ON DELETE CASCADE,
    date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, post_id)
);

-- R10: Comment Like Table
CREATE TABLE comment_like (
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE,
    post_id INT REFERENCES comment_post(id) ON DELETE CASCADE,
    date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, post_id)
);

-- R11: Question Answered Notification
CREATE TABLE question_answered_notification (
    id SERIAL PRIMARY KEY,
    user_receive_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN NOT NULL DEFAULT FALSE,
    user_trigger_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    question_id INT REFERENCES question_post(id) ON DELETE CASCADE NOT NULL,
```

```
    answer_id INT REFERENCES answer_post(id) ON DELETE CASCADE NOT NULL
);

-- R12: Question Liked Notification
CREATE TABLE question_liked_notification (
    id SERIAL PRIMARY KEY,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN NOT NULL DEFAULT FALSE,
    user_receive_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    user_trigger_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    question_id INT REFERENCES question_post(id) ON DELETE CASCADE NOT NULL
);

-- R13: Answer Commented Notification
CREATE TABLE answer_commented_notification (
    id SERIAL PRIMARY KEY,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN NOT NULL DEFAULT FALSE,
    user_receive_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    user_trigger_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    answer_id INT REFERENCES answer_post(id) ON DELETE CASCADE NOT NULL,
    comment_id INT REFERENCES comment_post(id) ON DELETE CASCADE NOT NULL
);

-- R14: Answer Liked Notification
CREATE TABLE answer_liked_notification (
    id SERIAL PRIMARY KEY,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN NOT NULL DEFAULT FALSE,
    user_receive_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    user_trigger_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    answer_id INT REFERENCES answer_post(id) ON DELETE CASCADE NOT NULL
);

-- R15: Comment Liked Notification
CREATE TABLE comment_liked_notification (
    id SERIAL PRIMARY KEY,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_read BOOLEAN NOT NULL DEFAULT FALSE,
    user_receive_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    user_trigger_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    comment_id INT REFERENCES comment_post(id) ON DELETE CASCADE NOT NULL
);

-- R16: User_Follow_Tag Table
CREATE TABLE user_follow_tag (
    user_id INT REFERENCES "user"(id) ON DELETE CASCADE NOT NULL,
    tag_id INT REFERENCES tag(id) ON DELETE CASCADE NOT NULL
);

-- Indexes
CREATE INDEX idx_question_tag ON question_post USING hash (tag_id);
CREATE INDEX idx_question_likes ON question_post (nr_likes);
CREATE INDEX idx_question_date ON question_post (date);

-- Full-Search Text Index
ALTER TABLE question_post
ADD COLUMN tsvectors TSVECTOR;

CREATE OR REPLACE FUNCTION question_title_search_update()
RETURNS TRIGGER AS $$
BEGIN
    NEW.tsvectors :=
        setweight(to_tsvector('english', NEW.title), 'A') ||
        setweight(to_tsvector('english', NEW.text), 'B');
    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_title_search_update_trigger
BEFORE INSERT OR UPDATE ON question_post
FOR EACH ROW
EXECUTE FUNCTION question_title_search_update();

CREATE INDEX idx_question_title_search ON question_post USING GIN (tsvectors);

-- Trigger 1: Update nr_likes on post when a new like is added
CREATE OR REPLACE FUNCTION update_post_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE question_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_like_trigger
AFTER INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION update_post_like();

-- Trigger 2: Prevent user from liking their own post
CREATE OR REPLACE FUNCTION prevent_self_like() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM question_post WHERE id = NEW.post_id) = NEW.user_id THEN
        RAISE EXCEPTION 'Cannot like own post';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_self_like_trigger
BEFORE INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like();

-- Trigger 3: Update nr_likes on post when a like is removed
CREATE OR REPLACE FUNCTION update_post_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE question_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_unlike_trigger
AFTER DELETE ON question_like
FOR EACH ROW
EXECUTE FUNCTION update_post_unlike();

-- Trigger 4: Update nr_likes on answer post when a new like is added
CREATE OR REPLACE FUNCTION update_answer_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE answer_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_like_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION update_answer_like();

-- Trigger 5: Prevent user from liking their own answer
CREATE OR REPLACE FUNCTION prevent_self_like_answer() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM answer_post WHERE id = NEW.post_id) = NEW.user_id THEN
```

```
        RAISE EXCEPTION 'Cannot like own answer';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_self_like_trigger
BEFORE INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like_answer();

-- Trigger 6: Update nr_likes on answer post when a like is removed
CREATE OR REPLACE FUNCTION update_answer_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE answer_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_unlike_trigger
AFTER DELETE ON answer_like
FOR EACH ROW
EXECUTE FUNCTION update_answer_unlike();

-- Trigger 7: Update nr_likes on comment post when a new like is added
CREATE OR REPLACE FUNCTION update_comment_like() RETURNS TRIGGER AS $$
BEGIN
    UPDATE comment_post SET nr_likes = nr_likes + 1 WHERE id = NEW.post_id;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_like_trigger
AFTER INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION update_comment_like();

-- Trigger 8: Prevent user from liking their own comment
CREATE OR REPLACE FUNCTION prevent_self_like_comment() RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT user_id FROM comment_post WHERE id = NEW.post_id) = NEW.user_id THEN
        RAISE EXCEPTION 'Cannot like own comment';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_self_like_trigger
BEFORE INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION prevent_self_like_comment();

-- Trigger 9: Update nr_likes on comment post when a like is removed
CREATE OR REPLACE FUNCTION update_comment_unlike() RETURNS TRIGGER AS $$
BEGIN
    UPDATE comment_post SET nr_likes = nr_likes - 1 WHERE id = OLD.post_id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_unlike_trigger
AFTER DELETE ON comment_like
FOR EACH ROW
EXECUTE FUNCTION update_comment_unlike();

-- Trigger 10: Create a notification when a question is answered
CREATE OR REPLACE FUNCTION notify_question_answered() RETURNS TRIGGER AS $$
```

```
DECLARE
    question_author INT;
BEGIN
    SELECT user_id INTO question_author
    FROM question_post
    WHERE id = NEW.question_id;

    IF question_author IS NOT NULL AND EXISTS (SELECT 1 FROM "user" WHERE id = NEW.user_id) THEN
        INSERT INTO question_answered_notification(user_receive_id, user_trigger_id, question_id, answer_id, date)
        VALUES (
            question_author,
            NEW.user_id,
            NEW.question_id,
            NEW.id,
            NOW()
        );
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_answered_notification_trigger
AFTER INSERT ON answer_post
FOR EACH ROW
EXECUTE FUNCTION notify_question_answered();

-- Trigger 11: Create a notification when a question is liked
CREATE OR REPLACE FUNCTION notify_question_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO question_liked_notification(user_receive_id, user_trigger_id, question_id)
    VALUES (
        (SELECT user_id FROM question_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER question_liked_notification_trigger
AFTER INSERT ON question_like
FOR EACH ROW
EXECUTE FUNCTION notify_question_liked();

-- Trigger 12: Create a notification when an answer is liked
CREATE OR REPLACE FUNCTION notify_answer_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO answer_liked_notification(user_receive_id, user_trigger_id, answer_id)
    VALUES (
        (SELECT user_id FROM answer_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_liked_notification_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION notify_answer_liked();

-- Trigger 13: Create a notification when a comment is liked
CREATE OR REPLACE FUNCTION notify_comment_liked() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO comment_liked_notification(user_receive_id, user_trigger_id, comment_id)
```

```
    VALUES (
        (SELECT user_id FROM comment_post WHERE id = NEW.post_id),
        NEW.user_id,
        NEW.post_id
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER comment_liked_notification_trigger
AFTER INSERT ON comment_like
FOR EACH ROW
EXECUTE FUNCTION notify_comment_liked();

-- Trigger 14: Create a notification when a comment is made on an answer
CREATE OR REPLACE FUNCTION notify_answer_commented() RETURNS TRIGGER AS $$
DECLARE
    answer_author INT;
BEGIN
    SELECT user_id INTO answer_author
    FROM answer_post
    WHERE id = NEW.answer_id;

    IF answer_author IS NOT NULL AND EXISTS (SELECT 1 FROM "user" WHERE id = NEW.user_id) THEN
        INSERT INTO answer_commented_notification(user_receive_id, user_trigger_id, answer_id, comment_id, date)
        VALUES (
            answer_author,
            NEW.user_id,
            NEW.answer_id,
            NEW.id,
            NOW()
        );
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER answer_commented_notification_trigger
AFTER INSERT ON comment_post
FOR EACH ROW
EXECUTE FUNCTION notify_answer_commented();

-- Trigger 15: Trigger for Anonymizing Users
CREATE OR REPLACE FUNCTION anonymize_user()
RETURNS TRIGGER AS $$
BEGIN
    -- Update user data to anonymized values instead of deleting
    UPDATE "user"
    SET
        name = 'Anonymous',
        tagname = 'Anonymous_' || OLD.id,
        email = NULL,
        pw = NULL,
        age = NULL,
        country = NULL,
        degree = NULL,
        icon = NULL,
        is_admin = FALSE,
        is_moderator = FALSE,
        is_banned = TRUE
    WHERE id = OLD.id;

    RETURN NULL;  -- Prevent actual DELETE operation
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trigger_anonymize_user
BEFORE DELETE ON "user"
FOR EACH ROW
EXECUTE FUNCTION anonymize_user();

-- Trigger 16: Trigger for the "Expert" Badge (100 Likes Across All Answers)
CREATE OR REPLACE FUNCTION assign_badge(p_user_id INT, p_badge_name TEXT) RETURNS VOID AS $$
DECLARE
    v_badge_id INT;
BEGIN
    -- Find the badge ID based on the badge name
    SELECT id INTO v_badge_id
    FROM badge
    WHERE title = p_badge_name;

    -- Insert the badge only if it doesn't already exist for the user
    IF NOT EXISTS (
        SELECT 1 FROM user_badge
        WHERE user_badge.user_id = p_user_id AND user_badge.badge_id = v_badge_id
    ) THEN
        INSERT INTO user_badge (user_id, badge_id) VALUES (p_user_id, v_badge_id);
    END IF;
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION check_expert_badge() RETURNS TRIGGER AS $$
DECLARE
    total_likes INT;
BEGIN
    -- Count total likes across all answers by the user
    SELECT COUNT(*) INTO total_likes
    FROM answer_like l
    JOIN answer_post a ON l.post_id = a.id
    WHERE a.user_id = NEW.user_id;

    -- Assign the Expert badge if total likes reach 100
    IF total_likes >= 100 THEN
        PERFORM assign_badge(NEW.user_id, 'Expert');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER expert_badge_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION check_expert_badge();

-- Trigger 17: Trigger for the "Contributor" Badge (First Question)
CREATE OR REPLACE FUNCTION check_contributor_badge() RETURNS TRIGGER AS $$
BEGIN
    -- Assign the Contributor badge if the user creates their first question
    PERFORM assign_badge(NEW.user_id, 'Contributor');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER contributor_badge_trigger
AFTER INSERT ON question_post
FOR EACH ROW
EXECUTE FUNCTION check_contributor_badge();

-- Trigger 18: Trigger for the "Top Answerer" Badge (50 Likes on a Single Answer)
CREATE OR REPLACE FUNCTION check_top_answerer_badge() RETURNS TRIGGER AS $$
DECLARE
    answer_likes INT;
```

```
BEGIN
    -- Count likes for the specific answer
    SELECT COUNT(*) INTO answer_likes
    FROM answer_like
    WHERE post_id = NEW.post_id;

    -- Assign the Top Answerer badge if likes reach 50 on this answer
    IF answer_likes >= 50 THEN
        PERFORM assign_badge(NEW.user_id, 'Top Answerer');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER top_answerer_badge_trigger
AFTER INSERT ON answer_like
FOR EACH ROW
EXECUTE FUNCTION check_top_answerer_badge();

-- Trigger 19: Trigger for the "Rising Star" Badge (10 Answers)
CREATE OR REPLACE FUNCTION check_rising_star_badge() RETURNS TRIGGER AS $$
DECLARE
    total_answers INT;
BEGIN
    -- Count total answers by the user
    SELECT COUNT(*) INTO total_answers
    FROM answer_post
    WHERE user_id = NEW.user_id;

    -- Assign the Rising Star badge if answers reach 10
    IF total_answers >= 10 THEN
        PERFORM assign_badge(NEW.user_id, 'Rising Star');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER rising_star_badge_trigger
AFTER INSERT ON answer_post
FOR EACH ROW EXECUTE FUNCTION check_rising_star_badge();

-- Trigger 20: Trigger for the "Community Leader" Badge (50 Posts)
CREATE OR REPLACE FUNCTION check_community_leader_badge() RETURNS TRIGGER AS $$
DECLARE
    total_posts INT;
BEGIN
    -- Count total posts by the user (questions + answers)
    SELECT
        (SELECT COUNT(*) FROM question_post WHERE user_id = NEW.user_id) +
        (SELECT COUNT(*) FROM answer_post WHERE user_id = NEW.user_id)
    INTO total_posts;

    -- Assign the Community Leader badge if total posts reach 50
    IF total_posts >= 50 THEN
        PERFORM assign_badge(NEW.user_id, 'Community Leader');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER community_leader_badge_trigger_question
AFTER INSERT ON question_post
FOR EACH ROW
EXECUTE FUNCTION check_community_leader_badge();
```

```
CREATE TRIGGER community_leader_badge_trigger_answer
AFTER INSERT ON answer_post
FOR EACH ROW
EXECUTE FUNCTION check_community_leader_badge();

-- Transaction 1: Insert answer if question exists
CREATE OR REPLACE FUNCTION AddAnswerIfQuestionExists(
    question_id INT,
    answer_content TEXT,
    user_id INT
)
RETURNS VOID AS $$
BEGIN
    BEGIN
        IF EXISTS (SELECT 1 FROM question_post WHERE id = question_id) THEN
            INSERT INTO answer_post(text, date, is_edited, user_id, nr_likes)
            VALUES (answer_content, NOW(), FALSE, user_id, 0);
        ELSE
            RAISE EXCEPTION 'Question does not exist';
        END IF;

    EXCEPTION
        WHEN OTHERS THEN
            RAISE EXCEPTION 'An error occurred while inserting the answer';
    END;
END;
$$ LANGUAGE plpgsql;

-- Transaction 2: Insert comment if both question and answer exist
CREATE OR REPLACE FUNCTION AddCommentIfQuestionAndAnswerExist(
    question_id INT,
    answer_id INT,
    comment_content TEXT,
    user_id INT
)
RETURNS VOID AS $$
BEGIN
    BEGIN
        IF EXISTS (SELECT 1 FROM question_post WHERE id = question_id)
           AND EXISTS (SELECT 1 FROM answer_post WHERE id = answer_id) THEN
            INSERT INTO comment_post(text, date, is_edited, user_id, nr_likes)
            VALUES (comment_content, NOW(), FALSE, user_id, 0);
        ELSE
            RAISE EXCEPTION 'Question or answer does not exist';
        END IF;

    EXCEPTION
        WHEN OTHERS THEN
            RAISE EXCEPTION 'An error occurred while inserting the comment';
    END;
END;
$$ LANGUAGE plpgsql;
```

A.2. Database population

```
-- Population
INSERT INTO "user" (user_id, user_name, user_tagname, user_email, user_pw, user_age, user_country, user_degree, user_ic
VALUES
(1, 'João Silva', 'joao.s', 'joao.silva@example.com', 'password123', 22, 'Portugal', 'Master', 'joao_icon.png', false,
(2, 'Mariana Costa', 'mariana.c', 'mariana.costa@example.com', 'password456', 23, 'Portugal', 'Bachelor', 'mariana_icon
(3, 'Francisco Santos', 'francisco.s', 'francisco.santos@example.com', 'password789', 25, 'Portugal', 'PhD', 'francisco_
(4, 'Inês Mendes', 'ines.m', 'ines.mendes@example.com', 'securepass', 21, 'Portugal', 'Bachelor', 'ines_icon.png', true
(5, 'Rui Ferreira', 'rui.f', 'rui.ferreira@example.com', 'mypassword', 24, 'Portugal', 'Master', 'rui_icon.png', false,
```

```
(6, 'Carla Oliveira', 'carla.o', 'carla.oliveira@example.com', 'carla1234', 20, 'Portugal', 'Bachelor', 'carla_icon.png
(7, 'André Silva', 'andre.s', 'andre.silva@example.com', 'andresilva', 28, 'Portugal', 'Master', 'andre_icon.png', false
```

Revision history

Changes made to the first submission:

> No changes were made yet!

`GROUP24042, 17/10/2024`

- Eduardo Cruz, up202205128@up.pt
- Jorge Mesquita, up202108614@up.pt
- Mariana Marques, up201606434@up.pt
- Tatiana Lin, up202206371@up.pt

Editor of Submission: Tatiana Lin

Comments