ebd

Last edited by **Duarte Manuel Gonçalves Marques** 3 weeks ago

EBD: Database Specification Component

Connectify aims to revolutionize online social interactions by establishing a new paradigm in social networking that prioritizes user empowerment, trust, and community. We envision a platform where individuals can engage authentically, build meaningful relationships, and share their lives in a safe environment, ultimately creating a more connected and supportive digital world. Our commitment to user autonomy and privacy not only sets us apart but also inspires a movement towards more responsible and user-focused online experiences.

A4: Conceptual Data Model

The artifact goal is to outline the main classes, attributes, and associations of the system's data model, clarifying core entities, relationships, and constraints and providing a foundation for consistent database and software development through the Class Diagram. Regarding the Additional Business Rules, their purpose is to define key operational constraints and validations and ensure the system's behavior aligns with business needs, supporting data integrity and application workflows.

1. Class diagram

The diagram in Figure 1 illustrates the primary organizational entities for the Connectify platform, showcasing relationships between them, attributes and their data types, as well as the multiplicity of associations.

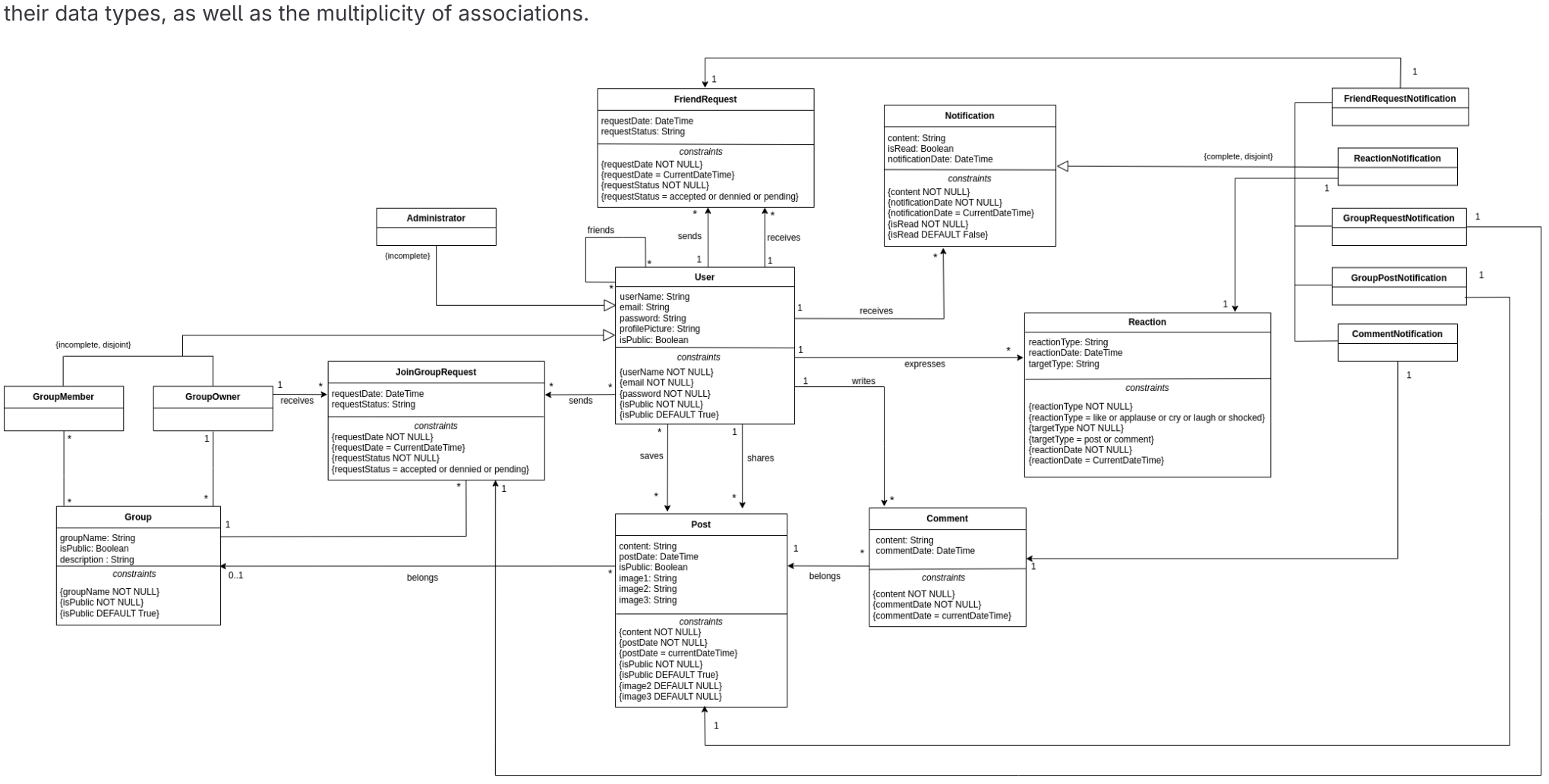

Figure 1: Connectify Class Diagram.

2. Additional Business Rules

- BR07. Authenticated users can only view basic public information (name and profile picture) of private profiles unless they are approved friends.
- BR08. A post must have content, either text, media or both.
- BR09. When a post is shared, the original post's privacy settings are inherited.
- BR10. Users can decide to turn notifications off.
- BR11. A user can only post in group they belong to.
- BR12. A user cannot request to follow themselves or someone they already follow.
- BR13. A user must be able to unfollow another user anytime they want.

A5: Relational Schema, validation and schema refinement

This artifact presents the Relational Schema, translating the conceptual model into a structured format suitable for database implementation. It includes validation steps to ensure data integrity and schema refinement to optimize performance, removing redundancies and enhancing efficiency.

1. Relational Schema

Relation Reference	Relation Compact Notation
R01	user(id **NN**, username **NN UK**, email **NN**, profilePicture, password **NN**, isPublic **DF** TRUE)

Relation Reference	Relation Compact Notation
R02	administrator(userId -> user **NN**)
R03	group(id **NN**, groupName **NN**, description, isPublic **DF** TRUE, ownerId -> user **NN**)
R04	post(id **NN**, content **NN**, image1 **DF** NULL, image2 **DF** NULL, image3 **DF** NULL, postDate **NN CK** postDate = currentDateTime, isPublic **DF** TRUE, groupId -> group, userId -> user **NN**)
R05	saved_post(userId -> user **NN**, postId -> post **NN**)
R06	comment(id **NN**, content **NN**, commentDate **NN CK** commentDate = currentDateTime, postId -> post **NN**, userId -> user **NN**)
R07	reaction(id **NN**, reactionType **NN CK** Type IN ReactionTypes, reactionDate **NN CK** reactionDate = CurrentDateTime, targetId **NN**, targetType **NN CK** Type IN TargetTypes, userId -> user **NN**)
R08	friend_request(id **NN**, requestDate **NN CK** requestDate = CurrentDateTime, requestStatus **NN CK** Status **IN** StatusFriendRequest, senderId -> user **NN**, receiverId -> user **NN**)
R09	friendship(userId1 -> user **NN**, userId2 -> user **NN**)
R10	join_group_request(id **NN**, requestDate **NN CK** requestDate = CurrentDateTime, requestStatus **NN CK** Status **IN** StatusGroupRequest, groupId -> group **NN**, userId -> user **NN**, groupOwnerId -> user **NN**)
R11	group_member(userId -> user **NN**, groupId -> group **NN**)
R12	group_owner(userId -> user **NN**, groupId -> group **NN**)
R13	notification(id **NN**, content **NN**, isRead **DF** FALSE, notificationDate **NN CK** notificationDate = CurrentDateTime, userId -> user **NN**)
R14	comment_notification(notificationId -> notification **NN**, commentId -> comment **NN**)
R15	reaction_notification(notificationId -> notification **NN**, reactionId -> reaction **NN**)
R16	friend_request_notification(notificationId -> notification **NN**, friendRequestId -> friend_request **NN**)
R17	group_request_notification(notificationId -> notification **NN**, groupRequestId -> join_group_request **NN**)
R18	group_post_notification(notificationId -> notification **NN**, postId -> post **NN**)

Legend:

- UK = UNIQUE KEY
- NN = NOT NULL
- DF = DEFAULT
- CK = CHECK.

2. Domains

Domain Name	Domain Specification
CurrentDateTime	DATETIME DEFAULT CURRENT_TIMESTAMP
Reactiontypes	ENUM ('like', 'laugh', 'cry', 'applause','shocked')
StatusGroupRequest	ENUM ('requested', 'accepted', 'denied')
StatusFriendRequest	ENUM ('requested', 'accepted', 'denied')
TargetTypes	ENUM ('post', 'comment')

3. Schema validation

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

TABLE R01	**user**
Keys	{id}, {email}, {userName}

TABLE R01	user
Functional Dependencies:	
FD0101	{id} -> {email, userName, password, profilePicture, isPublic}
FD0102	{email} -> {id, userName, password, profilePicture, isPublic}
FD0103	{userName} -> {id, email, password, profilePicture, isPublic}
NORMAL FORM	BCNF

TABLE R02	administrator
Keys	{id}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R03	post
Keys	{id}
Functional Dependencies:	
FD0201	{id} -> {content, postDate, isPublic, image1, image2, image3, groupId, userId}
NORMAL FORM	BCNF

TABLE R04	saved_post
Keys	{userId, postId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R05	comment
Keys	{id}
Functional Dependencies:	
FD0301	{id} -> {content, commentDate, postId, userId}
NORMAL FORM	BCNF

TABLE R06	reaction
Keys	{id}
Functional Dependencies:	
FD1001	{id} → {reactionType, reactionDate, targetId, targetType, userId}
NORMAL FORM	BCNF

TABLE R07	friend_request
Keys	{id}

TABLE R07	friend_request
Functional Dependencies:	
FD0801	{id} -> {requestDate, requestStatus, senderId, receiverId}
NORMAL FORM	BCNF

TABLE R08	friendship
Keys	{userId1, userId2}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R09	group
Keys	{id}
Functional Dependencies:	
FD0401	{id} -> {groupName, description, isPublic, ownerId}
NORMAL FORM	BCNF

TABLE R10	join_group_request
Keys	{id}
Functional Dependencies:	
FD0701	{id} -> {requestDate, requestStatus, groupId, userId, groupOwnerId}
NORMAL FORM	BCNF

TABLE R11	group_member
Keys	{userId, groupId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R12	group_owner
Keys	{userId, groupId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R13	notification
Keys	{id}
Functional Dependencies:	
FD0901	{id} -> {content, isRead, notificationDate, userId}
NORMAL FORM	BCNF

TABLE R14	comment_notification
Keys	{notificationId, commentId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R15	reaction_notification
Keys	{notificationId, reactionId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R16	friend_request_notification
Keys	{notificationId, friendRequestId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R17	group_request_notification
Keys	{notificationId, groupRequestId}
Functional Dependencies:	none
NORMAL FORM	BCNF

TABLE R18	group_post_notification
Keys	{notificationId, postId}
Functional Dependencies:	none
NORMAL FORM	BCNF

The schema is in BCNF as all functional dependencies are satisfied by superkeys, and no partial or transitive dependencies exist. Each table is fully normalized, with every non-key attribute depending solely on a candidate key, ensuring no further modifications are required to meet BCNF criteria.

A6: Indexes, triggers, transactions and database population

The artifact aims to optimize database performance and integrity through efficient indexing, triggers for automation and rule enforcement, and transaction management for reliable data operations.

1. Database Workload

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	user	10k	100/ day
R02	administrator	10	10
R03	post	10k	100/day
R04	saved_post	1k	100/day
R05	comment	10k	100/ day
R06	reaction	10k	100/ day

Relation reference	Relation Name	Order of magnitude	Estimated growth
R07	friend_request	1k	100/ day
R08	friendship	1k	100/ day
R09	group	100	10 / day
R10	join_group_request	100	10 / day
R11	group_member	1k	100/ day
R12	group_owner	100	10/day
R13	notification	10k	100/day
R14	comment_notification	100	100/day
R15	reaction_notification	100	100/day
R16	friend_request_notification	100	100/day
R17	group_request_notification	100	100/day
R18	group_post_notification	100	100/day

2. Proposed Indices

2.1. Performance Indices

Index	IDX01
Relation	post
Attribute	user_id
Type	B-tree
Cardinality	High
Clustering	Yes
Justification	The Post relation is expected to grow significantly as users create more posts. Indexing the user_id attribute will allow for faster retrieval of posts by individual users, which is a common query pattern in social platforms. Since the user_id column will have a moderate number of distinct values and frequent exact match queries, a B-tree index is suitable. Clustering will optimize data storage based on user_id, improving access times for user-specific queries.
`SQL code`	`CREATE INDEX idx_user_posts ON post(user_id);` `CLUSTER Post USING idx_user_posts;`

Index	IDX02
Relation	post
Attribute	postDate
Type	B-tree
Cardinality	High
Clustering	No

Index	IDX02
Justification	This index is essential for queries that retrieve posts sorted by postDate, allowing the application to display the most recent posts first. Since users often access their feeds or timelines based on date filters, this index significantly enhances retrieval speed for those queries. Clustering is not applied here because frequent updates and deletions of posts could lead to significant overhead in physically reorganizing the data. The B-tree index structure efficiently supports fast lookups without the need for the data to be physically ordered on disk, maintaining performance while minimizing maintenance costs during updates.
`SQL code`	`CREATE INDEX idx_post_postdate ON post(post_date);`

Index	IDX03
Relation	notification
Attribute	userId, notificationDate
Type	B-tree (Composite Index)
Cardinality	Medium
Clustering	No
Justification	The notification table has high probability of being large, since there are a lot of different types of notifications for each user. Several queries needs to access the notification table of it's user. Creating this index will allow us to filter the notifications of one user faster. Since this table has a high update rate, we chose not apply cluster to this index.
`SQL code`	`CREATE INDEX idx_notification_user_date ON notification (user_id, notification_date);`

2.2. Full-text Search Indices

Index	IDX04
Relation	user
Attribute	username
Type	GIN
Clustering	No
Justification	This index supports full-text search functionality for finding users by their username. It improves search performance, enabling faster results when users look for other members. The GIN index is appropriate due to the nature of the text data, which is read frequently but updated infrequently.

SQL Code

```
ALTER TABLE user_ ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION user_search_update() RETURNS TRIGGER AS $$

BEGIN

   IF TG_OP = 'INSERT' THEN

        NEW.tsvectors := to_tsvector('portuguese', NEW.username);

   END IF;

   IF TG_OP = 'UPDATE' THEN

        IF (NEW.username <> OLD.username) THEN

            NEW.tsvectors := to_tsvector('portuguese', NEW.username);

        END IF;

   END IF;

   RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER user_search_update

BEFORE INSERT OR UPDATE ON user_

FOR EACH ROW

EXECUTE PROCEDURE user_search_update();

CREATE INDEX idx_user_search ON user_ USING GIN (tsvectors);
```

Index	**IDX05**
Relation	post
Attribute	content
Type	GIN
Clustering	No
Justification	This index is created to facilitate full-text search on post content, which is crucial for users to search posts based on keywords or phrases. The GIN index type is selected due to its efficiency in handling large text-based columns.

SQL Code

```
ALTER TABLE post

ADD COLUMN IF NOT EXISTS tsvectors TSVECTOR;

CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$

BEGIN

   IF TG_OP = 'INSERT' THEN

       NEW.tsvectors := to_tsvector('portuguese', NEW.content);

   ELSIF TG_OP = 'UPDATE' THEN

       IF NEW.content <> OLD.content THEN

           NEW.tsvectors := to_tsvector('portuguese', NEW.content);

       END IF;

   END IF;

   RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER update_post_tsvectors

BEFORE INSERT OR UPDATE ON post

FOR EACH ROW

EXECUTE FUNCTION post_search_update();

CREATE INDEX idx_post_content ON post USING GIN (tsvectors);
```

Index	**IDX06**
Relation	group
Attribute	groupName, description
Type	GIN
Clustering	No
Justification	This index is designed to enable full-text search on group descriptions, allowing users to find groups that match certain keywords. This is essential for discovering groups based on their descriptions. The GIN index type is chosen as it supports full-text search effectively and performs well for read-heavy scenarios, which suits the group entity where updates are less frequent.

SQL Code

```
ALTER TABLE group_ ADD COLUMN IF NOT EXISTS tsvectors TSVECTOR;

CREATE FUNCTION group_search_update() RETURNS TRIGGER AS $$

BEGIN

   IF TG_OP = 'INSERT' THEN

       NEW.tsvectors := ( setweight(to_tsvector('portuguese', NEW.group_name), 'A') ||
setweight(to_tsvector('portuguese', NEW.description), 'B') );

   ELSIF TG_OP = 'UPDATE' THEN

        IF NEW.group_name <> OLD.group_name OR NEW.description <> OLD.description THEN

           NEW.tsvectors := ( setweight(to_tsvector('portuguese', NEW.group_name), 'A') ||
setweight(to_tsvector('portuguese', NEW.description), 'B') );

       END IF;

   END IF;

   RETURN NEW;

END;

$$ LANGUAGE plpgsql;

CREATE TRIGGER group_search_update

BEFORE INSERT OR UPDATE ON group_

FOR EACH ROW

EXECUTE FUNCTION group_search_update();

CREATE INDEX idx_group_description ON group_ USING GIN (tsvectors);
```

3. Triggers

Trigger	TRIGGER01
Description	This trigger references BR02 and ensures that users cannot send duplicate friendship requests if one is already pending approval.

SQL Code	`CREATE OR REPLACE FUNCTION enforce_friend_request_limit()`
	`RETURNS TRIGGER AS $$`
	`BEGIN`
	`IF EXISTS ( SELECT 1 FROM friend_request WHERE senderId = NEW.senderId AND receiverId = NEW.receiverId AND requestStatus NOT IN ('denied') ) THEN`
	`RAISE EXCEPTION 'Not successful: cannot send more than one friend request to the same user.';`
	`END IF;`
	`RETURN NEW;`
	`END;`
	`$$ LANGUAGE plpgsql;`
	`CREATE TRIGGER trg_enforce_friend_request_limit`
	`BEFORE INSERT ON friend_request`
	`FOR EACH ROW`
	`EXECUTE FUNCTION enforce_friend_request_limit();`

Trigger	TRIGGER02
Description	This trigger references BR04 and prevents users from adding multiple reactions to the same post, allowing only one reaction at a time.

SQL Code	`CREATE OR REPLACE FUNCTION enforce_reaction_limit() RETURNS TRIGGER AS $$`
	`BEGIN`
	`  IF EXISTS ( SELECT 1 FROM reaction WHERE targetId = NEW.targetId AND targetType = NEW.targetType AND user_id = NEW.user_id) THEN`
	`  RAISE EXCEPTION 'User already reacted to this post.';`
	`  END IF;`
	`  RETURN NEW;`
	`END;`
	`$$ LANGUAGE plpgsql;`
	`CREATE TRIGGER trg_enforce_reaction_limit`
	`BEFORE INSERT ON reaction`
	`FOR EACH ROW`
	`EXECUTE FUNCTION enforce_reaction_limit();`

Trigger	**TRIGGER03**
Description	This trigger references BR08 and ensures that each post has content, whether text, media, or both.

SQL Code	`CREATE OR REPLACE FUNCTION enforce_post_content() RETURNS TRIGGER AS $$`
	`BEGIN`
	`  IF NEW.content IS NULL AND NEW.image1 IS NULL AND NEW.image2 IS NULL AND NEW.image3 IS NULL`
	`  THEN RAISE EXCEPTION 'A post must contain text or at least one image.';`
	`  END IF;`
	`  RETURN NEW;`
	`END;`
	`$$ LANGUAGE plpgsql;`
	`CREATE TRIGGER trg_enforce_post_content`
	`BEFORE INSERT OR UPDATE ON post`
	`FOR EACH ROW`
	`EXECUTE FUNCTION enforce_post_content();`

Trigger	**TRIGGER04**
Description	This trigger references BR05 and ensures that user data is anonymized upon account deletion, while retaining content like comments, likes, and shared data.

SQL Code	`CREATE OR REPLACE FUNCTION anonymize_user_data()`
	`RETURNS TRIGGER AS $$`
	`BEGIN`
	`  UPDATE comment_ SET user_id = 0 WHERE user_id = OLD.id;`
	`  UPDATE reaction SET user_id = 0 WHERE user_id = OLD.id;`
	`  UPDATE post SET user_id = 0 WHERE user_id = OLD.id;`
	`  RETURN OLD;`
	`END;`
	`$$ LANGUAGE plpgsql;`
	`CREATE TRIGGER trg_anonymize_user_data`
	`AFTER DELETE ON users`
	`FOR EACH ROW`
	`EXECUTE FUNCTION anonymize_user_data();`

Trigger	TRIGGER05
Description	This trigger references BR11 and enforces that users can only post in groups they belong to.

SQL Code	`CREATE OR REPLACE FUNCTION enforce_group_posting() RETURNS TRIGGER AS $$`
	`BEGIN`
	`  IF NOT EXISTS ( SELECT 1 FROM group_member WHERE groupId = NEW.groupId AND userId = NEW.userId ) THEN`
	`  RAISE EXCEPTION 'User must be a member of the group to post.';`
	`  END IF;`
	`  RETURN NEW;`
	`END;`
	`$$ LANGUAGE plpgsql;`
	`CREATE TRIGGER trg_enforce_group_posting`
	`BEFORE INSERT ON post`
	`FOR EACH ROW`
	`EXECUTE FUNCTION enforce_group_posting();`

Trigger	TRIGGER06
Description	This trigger references BR03 and ensures that a group owner's approval is required for joining private groups.

SQL Code

```
CREATE OR REPLACE FUNCTION enforce_group_membership_control() RETURNS TRIGGER AS $$
BEGIN
  IF (SELECT isPublic FROM group_ WHERE id = NEW.groupId) = FALSE THEN
       IF NEW.requestStatus = 'Pending' THEN
           RAISE EXCEPTION 'Group membership requires owner approval.';
       END IF;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER trg_enforce_group_membership_control
BEFORE INSERT OR UPDATE ON join_group_request
FOR EACH ROW
EXECUTE FUNCTION enforce_group_membership_control();
```

Trigger	TRIGGER07
Description	This trigger was created on the development of the reactions in the final product and validates the reaction's target.

SQL Code

```
CREATE OR REPLACE FUNCTION validate_targeT_id()
RETURNS TRIGGER AS $$
BEGIN
  IF (NEW.target_type = 'post') THEN
    IF NOT EXISTS (SELECT 1 FROM post WHERE id=NEW.target_id) THEN
      RAISE EXCEPTION 'Target ID % not found in post table.', NEW.target_id;
    END IF;
  ELSEIF (NEW.target_type = 'comment') THEN
    IF NOT EXISTS (SELECT 1 FROM comment_ WHERE id=NEW.target_id) THEN
      RAISE EXCEPTION 'Target ID % not found in comment table.', NEW.target_id;
    END IF;
  ELSE
    RAISE EXCEPTION 'Invalid target type: %', NEW.target_type;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER trg_validate_target_id
BEFORE INSERT OR UPDATE ON reaction
FOR EACH ROW
EXECUTE FUNCTION validate_target_id();
```

4. Transactions

Transaction	TRAN01
Description	Create a new post
Justification	When a user creates a new post, it's crucial that the operation completes successfully and the data is consistent. If multiple users are posting simultaneously, the transaction ensures that each post is stored accurately without data corruption.
Isolation level	Serializable

`SQL CODE`	`CREATE OR REPLACE FUNCTION inserir_post(content TEXT, visibility BOOLEAN) RETURNS VOID AS $$`
	`BEGIN`
	`  INSERT INTO post (content, is_public, user_id) VALUES ($1, $2, COALESCE(current_user_id, 1));`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	TRAN02
Description	Create a new comment on a post and generates notification
Justification	When a user adds a comment, it is crucial to ensure that the comment is accurately associated with the correct post and user. This transaction maintains data integrity by using a serializable isolation level to prevent data corruption from concurrent operations. It ensures that if multiple users are commenting on the same post simultaneously, each comment is correctly linked to its respective user and post without any risk of inconsistent data being written to the database. Additionally, by generating a notification for the post owner, the system keeps them informed about interactions on their content, enhancing user engagement and ensuring a responsive user experience.
Isolation level	Serializable

`SQL Code`	`CREATE OR REPLACE FUNCTION add_comment( postId INT, userId INT, commentContent TEXT ) RETURNS VOID AS $$`
	`DECLARE`
	`  postOwnerId INT;`
	`  commentId INT;`
	`BEGIN`
	`  SELECT user_id INTO postOwnerId FROM post WHERE post_id = postId;`
	`  IF postOwnerId IS NULL THEN`
	`  RAISE EXCEPTION 'No valid post found.';`
	`  END IF;`
	`  INSERT INTO comment_ (post_id, user_id, comment_content, commentDate) VALUES (postId, userId, commentContent, NOW()) RETURNING id INTO commentId;`
	`  INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' \|\| userId \|\| ' commented on your post.', FALSE, NOW(), postOwnerId);`
	`  INSERT INTO comment_notification (notification_id, comment_id) VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), commentId);`
	`  END IF;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	TRAN03
Description	User reacts to post and generates notification

Justification	When a user reacts to a post, it is essential to create a corresponding notification entry in the `notification` table. Using a transaction ensures that if there's an issue during the notification creation (e.g., database error), the system can handle it gracefully, preventing inconsistent states where a reaction is recorded, but the notification is not created.
Isolation level	Serializable

`SQL Code`	`CREATE OR REPLACE FUNCTION add_reaction(postId INT, userId INT, reactionType TEXT) RETURNS VOID AS $$`
	`DECLARE`
	`  postOwnerId INT;`
	`  reactionId INT;`
	`BEGIN`
	`  SELECT user_id INTO postOwnerId FROM post WHERE post_id = postId;`
	`  INSERT INTO reaction (reactionType, reaction_date, post_id, user_id) VALUES (reactionType, NOW(), postId, userId) RETURNING reaction_id INTO reactionId;`
	`  INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' \|\| userId \|\| ' reacted to your post with ' \|\| reactionType, FALSE, NOW(), postOwnerId);`
	`  INSERT INTO reaction_notification (notification_id, reaction_id) VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), reactionId);`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN04**
Description	Update a user's profile information.
Justification	When updating profile data, it is vital to maintain data integrity. This transaction ensures that if an error occurs during the update process, no partial updates are made, preserving the consistency of user data. By using a transaction, we can ensure that the changes are fully applied or not at all, thus avoiding corrupt or incomplete user profiles.
Isolation level	Serializable

`SQL Code`	`CREATE OR REPLACE FUNCTION update_user_info(userId INT, newName TEXT, newEmail TEXT, newProfilePicture TEXT) RETURNS VOID AS $$`
	`BEGIN`
	`  UPDATE user_ SET username = newName, email = newEmail, profile_picture = newProfilePicture WHERE user_id = userId;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN05**
Description	Delete a post.
Justification	When a post is deleted, it is crucial to ensure that all related data (e.g., comments and likes) is also removed appropriately, preventing orphaned records and maintaining data integrity.
Isolation level	Serializable

SQL Code	`CREATE OR REPLACE FUNCTION delete_post(postId INT) RETURNS VOID AS $$`
	`BEGIN`
	`  DELETE FROM comment_ WHERE post_id = postId;`
	`  DELETE FROM reaction WHERE post_id = postId;`
	`  DELETE FROM post WHERE post_id = postId;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN06**
Description	Send friend request and generates notification
Justification	When a user sends a friend request, it is essential to ensure that only one pending request exists between two users at any time to prevent duplicate requests and maintain a clean user experience. This transaction guarantees data integrity by enforcing constraints on the friendship request process, particularly in environments with high concurrency where multiple users might send requests simultaneously. By using a transaction, we can ensure that the system checks for any existing pending requests before inserting a new one, thereby avoiding data corruption and ensuring that notifications are appropriately generated for both the sender and receiver. This safeguards against potential race conditions that could arise if requests were processed without proper transaction management.
Isolation level	Serializable

SQL Code	`CREATE OR REPLACE FUNCTION send_friend_request(sender_id INT, receiver_id INT) RETURNS VOID AS $$`
	`DECLARE`
	`  notification_id INT;`
	`BEGIN`
	`  INSERT INTO friend_request (request_date, request_status, sender_id, receiver_id) VALUES (NOW(), 'pending', sender_id, receiver_id);`
	`  INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' \|\| sender_id \|\| ' sent you a friend request.', FALSE, NOW(), receiver_id);`
	`  notification_id := currval(pg_get_serial_sequence('notification', 'notification_id'));`
	`  INSERT INTO friend_request_notification (notification_id, friend_request_id) VALUES (notification_id, (SELECT MAX(request_id) FROM friend_request WHERE sender_id = sender_id AND receiver_id = receiver_id));`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN07**
Description	Accept friend request
Justification	Accepting a friend request involves updating the request status and creating a friendship record. This transaction ensures that both actions are atomic, preventing partial updates if an error occurs.
Isolation level	Serializable

SQL Code	`CREATE OR REPLACE FUNCTION accept_friend_request(request_id INT, user_id1 INT, user_id2 INT) RETURNS VOID AS $$`
	`BEGIN`
	`UPDATE friend_request SET request_status = 'accepted' WHERE id = request_id;`
	`INSERT INTO friendship (userId1, userId2) VALUES (user_id1, user_id2);`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN08**
Description	Create new group
Justification	Creating a group requires inserting a new record into the `group` table, and it's essential that this operation completes successfully to maintain data integrity. If a failure occurs (e.g., due to a database error), it could result in an inconsistent state where no group is created but resources are reserved. A transaction ensures that all steps are completed successfully or none are applied, preserving data integrity.
Isolation level	Serializable

SQL Code	`CREATE OR REPLACE FUNCTION create_group(group_name TEXT, description TEXT, is_public BOOLEAN, owner_id INT) RETURNS VOID AS $$`
	`BEGIN`
	`INSERT INTO "group" (groupName, description, isPublic, ownerId) VALUES (group_name, description, is_public, owner_id);`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN09**
Description	Delete user
Justification	When a user is deleted, it is important to keep their shared data (comments, reviews, likes) for historical context while ensuring the data is anonymized to protect the user's identity. This transaction allows for the preservation of user-generated content without compromising user privacy.
Isolation level	Serializable

SQL Code	`CREATE OR REPLACE FUNCTION delete_user(user_id INT) RETURNS VOID AS $$`
	`BEGIN`
	`UPDATE comment SET userId = NULL, content = CONCAT('Deleted User: ', content) WHERE userId = user_id;`
	`UPDATE post SET userId = NULL, content = CONCAT('Deleted User Post: ', content) WHERE userId = user_id;`
	`UPDATE saved_post SET userId = NULL WHERE userId = user_id;`
	`UPDATE friend_request SET senderId = NULL WHERE senderId = user_id;`
	`UPDATE friend_request SET receiverId = NULL WHERE receiverId = user_id;`
	`UPDATE group_member SET user_id = NULL WHERE user_id = user_id;`
	`UPDATE group_owner SET user_id = NULL WHERE user_id = user_id;`
	`DELETE FROM "user" WHERE id = user_id;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN10**
Description	Save post
Justification	Saving a post is a critical action that updates the user's saved posts. If there's an error during the insertion (e.g., database connection issue or duplicate entry), it could result in an incomplete operation. Using a transaction ensures that the save action either completes entirely or not at all, maintaining the integrity of the user's saved posts list.
Isolation level	Serializable

`SQL Code`	`CREATE OR REPLACE FUNCTION save_post(user_id INT, post_id INT) RETURNS VOID AS $$`
	`BEGIN`
	`  INSERT INTO saved_post (user_id, postId) VALUES (user_id, post_id);`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN11**
Description	Remove post from saved posts
Justification	Removing a saved post involves deleting a record from the `saved_post` table. If an error occurs during this operation (e.g., the record does not exist or there's a database error), a transaction ensures that you can handle it properly and maintain data integrity. It guarantees that if any part of the operation fails, the system remains unchanged.
Isolation level	Serializable

`SQL Code`	`CREATE OR REPLACE FUNCTION remove_saved_post(user_id INT, post_id INT) RETURNS VOID AS $$`
	`BEGIN`
	`  DELETE FROM saved_post WHERE user_id = user_id AND postId = post_id;`
	`END;`
	`$$ LANGUAGE plpgsql;`

Transaction	**TRAN12**
Description	Send group request and generates notification
Justification	When a user sends a request to join a group, it is crucial to ensure that the request is recorded accurately to prevent duplicate requests. This transaction maintains data integrity by checking for existing requests and, upon successful insertion, generates a notification for the group owner. This informs them of the new request, enhancing engagement while ensuring that the system remains consistent even in cases of failure.
Isolation level	Serializable

SQL Code

```
CREATE OR REPLACE FUNCTION request_to_join_group(group_id INT, user_id INT) RETURNS VOID AS $$

DECLARE

   groupOwnerId INT;

   notification_id INT;

BEGIN

    INSERT INTO join_group_request (group_id, user_id, request_status, requested_at) VALUES (group_id, user_id, 'pending', NOW());

    SELECT owner_id INTO groupOwnerId FROM group_ WHERE group_id = group_id;

    INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' || user_id || ' has requested to join your group.', FALSE, NOW(), groupOwnerId);

    notification_id := LASTVAL();

    INSERT INTO group_request_notification (notification_id, group_request_id) VALUES (notification_id, (SELECT MAX(request_id) FROM join_group_request WHERE group_id = group_id AND user_id = user_id));

END;

$$ LANGUAGE plpgsql;
```

Transaction	**TRAN13**
Description	New post on group and generates notification
Justification	When a user creates a new post in a group, it is essential to ensure that the post is accurately recorded and linked to the correct group and user. This transaction guarantees that all operations are executed successfully; if any step fails (e.g., due to a database error), it prevents partial data from being saved, which could lead to inconsistencies. Additionally, generating notifications for all group members and the group owner ensures that they are informed about the new content, fostering engagement while maintaining data integrity throughout the process.
Isolation level	Serializable

SQL Code

```
CREATE OR REPLACE FUNCTION add_post( p_user_id INT, p_group_id INT, p_content TEXT ) RETURNS VOID AS $$

DECLARE

    groupOwnerId INT;

   groupMemberIds INT[];

   new_post_id INT;

   member_id INT;

BEGIN

    INSERT INTO post (user_id, group_id, content, post_date) VALUES (p_user_id, p_group_id, p_content, NOW()) RETURNING post_id INTO new_post_id;

    SELECT owner_id INTO groupOwnerId FROM group_ WHERE group_id = p_group_id;

    SELECT ARRAY_AGG(user_id) INTO groupMemberIds FROM group_member WHERE group_id = p_group_id;

    INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' || p_user_id || ' has posted in your group.', FALSE, NOW(), groupOwnerId);

   FOREACH member_id IN ARRAY groupMemberIds

   LOOP

        INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' || p_user_id || ' has posted in the group.', FALSE, NOW(), member_id);

        INSERT INTO group_post_notification (notification_id, post_id) VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), new_post_id);

    END LOOP;

END

$$ LANGUAGE plpgsql;
```

Transaction	TRAN14
Description	Accepts group request and generates notification
Justification	When a group join request is accepted, it is crucial to ensure that the acceptance is accurately recorded and linked to both the user and the group. This transaction guarantees that if any step fails (e.g., due to a database error), the operation does not leave the system in an inconsistent state. Additionally, generating notifications for the user whose request was accepted ensures they are informed, maintaining engagement while preserving data integrity throughout the process.
Isolation level	Serializable

SQL Code

```
CREATE OR REPLACE FUNCTION accept_join_group_request( p_request_id INT -- Declare request_id as a parameter )
RETURNS VOID AS $$

DECLARE

   requesterId INT;

   groupId INT;

    groupOwnerId INT;
   notification_id INT;

BEGIN

    SELECT user_id, group_id INTO requesterId, groupId FROM join_group_request WHERE request_id = p_request_id;

    UPDATE join_group_request SET request_status = 'accepted' WHERE request_id = p_request_id;

    INSERT INTO group_member (user_id, group_id) VALUES (requesterId, groupId);

    SELECT owner_id INTO groupOwnerId FROM group_ WHERE group_id = groupId;

    INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('Your request to join the
group has been accepted.', FALSE, NOW(), requesterId);

    INSERT INTO notification (content, is_read, notification_date, user_id) VALUES ('User ' || requesterId || '
has joined your group.', FALSE, NOW(), groupOwnerId);

    notification_id := currval(pg_get_serial_sequence('notification', 'notification_id'));

    INSERT INTO group_request_notification (notification_id, request_id) VALUES (notification_id, p_request_id);

END

$$ LANGUAGE plpgsql;
```

Annex A. SQL Code

Link for the Database schema: SQLScript.sql

Link for the Database population: populate.sql

A.1. Database schema

```sql
DROP SCHEMA IF EXISTS lbaw2453 CASCADE;
CREATE SCHEMA lbaw2453;
SET search_path = lbaw2453;


DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS administrator CASCADE;
DROP TABLE IF EXISTS post CASCADE;
DROP TABLE IF EXISTS saved_post CASCADE;
DROP TABLE IF EXISTS comment_ CASCADE;
DROP TABLE IF EXISTS reaction CASCADE;
DROP TABLE IF EXISTS friend_request CASCADE;
DROP TABLE IF EXISTS friendship CASCADE;
DROP TABLE IF EXISTS group_ CASCADE;
DROP TABLE IF EXISTS join_group_request CASCADE;
DROP TABLE IF EXISTS group_member CASCADE;
DROP TABLE IF EXISTS group_owner CASCADE;
DROP TABLE IF EXISTS notification CASCADE;
DROP TABLE IF EXISTS comment_notification CASCADE;
DROP TABLE IF EXISTS reaction_notification CASCADE;
DROP TABLE IF EXISTS friend_request_notification CASCADE;
DROP TABLE IF EXISTS group_request_notification CASCADE;
DROP TABLE IF EXISTS group_post_notification CASCADE;

DROP TYPE IF EXISTS statusGroup_request CASCADE;
DROP TYPE IF EXISTS statusFriendship_request CASCADE;
DROP TYPE IF EXISTS reactionType CASCADE;
```

```
-- Types

CREATE TYPE statusGroup_request AS ENUM ('pending', 'accepted', 'denied');
CREATE TYPE statusFriendship_request AS ENUM ('pending', 'accepted', 'denied');
CREATE TYPE reactionType AS ENUM ('like', 'laugh', 'cry', 'applause', 'shocked');
CREATE TYPE targetType AS ENUM('post', 'comment');

-- Tables

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username TEXT UNIQUE NOT NULL,
    email TEXT UNIQUE NOT NULL,
    profile_picture TEXT DEFAULT 'default.png',
    password TEXT NOT NULL,
    is_public BOOLEAN DEFAULT TRUE NOT NULL
);

CREATE TABLE administrator (
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    PRIMARY KEY (user_id)
);

CREATE TABLE group_ (
    id SERIAL PRIMARY KEY,
    owner_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    group_name TEXT NOT NULL,
    description TEXT,
    visibility BOOLEAN NOT NULL,
    is_public BOOLEAN DEFAULT TRUE NOT NULL
);

CREATE TABLE post (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    group_id INT REFERENCES group_(id) ON UPDATE CASCADE,
    content TEXT,
    IMAGE1 TEXT,
    IMAGE2 TEXT,
    IMAGE3 TEXT,
    is_public BOOLEAN DEFAULT TRUE NOT NULL,
    post_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CHECK ((content IS NOT NULL OR IMAGE1 IS NOT NULL) OR group_id IS NULL)
);

CREATE TABLE saved_post (
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    post_id INT NOT NULL REFERENCES post(id) ON UPDATE CASCADE,
    PRIMARY KEY (user_id, post_id)
);

CREATE TABLE comment_ (
    id SERIAL PRIMARY KEY,
    post_id INT NOT NULL REFERENCES post(id) ON UPDATE CASCADE,
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    comment_content TEXT NOT NULL,
    commentDate TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE reaction (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    target_id INT NOT NULL, -- ID de um post ou comentário
    target_type targetType NOT NULL, --  'post' ou 'comment'
    reaction_type reactionType NOT NULL,
```

```
    reaction_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


CREATE TABLE friend_request (
    id SERIAL PRIMARY KEY,
    request_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    request_status statusFriendship_request NOT NULL,
    sender_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    receiver_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE
);

CREATE TABLE friendship (
    user_id1 INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    user_id2 INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    PRIMARY KEY (user_id1, user_id2)
);

CREATE TABLE join_group_request (
    id SERIAL PRIMARY KEY,
    group_id INT NOT NULL REFERENCES group_(id) ON UPDATE CASCADE,
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    request_status statusGroup_request NOT NULL,
    requested_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE group_member (
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    group_id INT NOT NULL REFERENCES group_(id) ON UPDATE CASCADE,
    PRIMARY KEY (user_id, group_id)
);

CREATE TABLE group_owner (
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE,
    group_id INT NOT NULL REFERENCES group_(id) ON UPDATE CASCADE,
    PRIMARY KEY (user_id, group_id)
);

CREATE TABLE notification (
    id SERIAL PRIMARY KEY,
    content TEXT NOT NULL,
    is_read BOOLEAN DEFAULT false,
    notification_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    user_id INT NOT NULL REFERENCES users(id) ON UPDATE CASCADE
);


CREATE TABLE comment_notification (
    notification_id INT PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE,
    comment_id INT NOT NULL REFERENCES comment_(id) ON UPDATE CASCADE
);

CREATE TABLE reaction_notification (
    notification_id INT PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE,
    reaction_id INT NOT NULL REFERENCES reaction(id) ON UPDATE CASCADE
);

CREATE TABLE friend_request_notification (
    notification_id INT PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE,
    friend_request_id INT NOT NULL REFERENCES friend_request(id) ON UPDATE CASCADE
);

CREATE TABLE group_request_notification (
    notification_id INT PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE,
    group_request_id INT NOT NULL REFERENCES join_group_request(id) ON UPDATE CASCADE
);

CREATE TABLE group_post_notification (
```

```
    notification_id INT PRIMARY KEY REFERENCES notification(id) ON UPDATE CASCADE,
    post_id INT NOT NULL REFERENCES post(id) ON UPDATE CASCADE
);

-- Perfomance Indices

CREATE INDEX idx_user_posts ON post(user_id);
CLUSTER Post USING idx_user_posts;

CREATE INDEX idx_post_postdate ON post(post_date);
CREATE INDEX idx_notification_user_date ON notification (user_id, notification_date);

-- Full-text Search Indices

-- User FTS index
DROP FUNCTION IF EXISTS user_search_update() CASCADE;
ALTER TABLE users
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION user_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors := to_tsvector('portuguese', NEW.username);
    END IF;
    IF TG_OP = 'UPDATE' THEN
        IF (NEW.username <> OLD.username) THEN
            NEW.tsvectors := to_tsvector('portuguese', NEW.username);
        END IF;
    END IF;
 RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER user_search_update
BEFORE INSERT OR UPDATE ON users
FOR EACH ROW
EXECUTE PROCEDURE user_search_update();

CREATE INDEX idx_user_search ON users USING GIN (tsvectors);

-- Post FTS index
DROP FUNCTION IF EXISTS post_search_update() CASCADE;

ALTER TABLE post
ADD COLUMN IF NOT EXISTS tsvectors TSVECTOR;

CREATE FUNCTION post_search_update() RETURNS TRIGGER AS $$
BEGIN
  IF TG_OP = 'INSERT' THEN
    NEW.tsvectors := to_tsvector('portuguese', NEW.content);
  ELSIF TG_OP = 'UPDATE' THEN
    IF NEW.content <> OLD.content THEN
      NEW.tsvectors := to_tsvector('portuguese', NEW.content);
    END IF;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_post_tsvectors
BEFORE INSERT OR UPDATE ON post
FOR EACH ROW
EXECUTE FUNCTION post_search_update();

CREATE INDEX idx_post_content ON post USING GIN (tsvectors);
```

```
-- Group FTS index
DROP FUNCTION IF EXISTS group_search_update() CASCADE;

ALTER TABLE group_
ADD COLUMN IF NOT EXISTS tsvectors TSVECTOR;

CREATE FUNCTION group_search_update() RETURNS TRIGGER AS $$
BEGIN
    IF TG_OP = 'INSERT' THEN
        NEW.tsvectors := (
            setweight(to_tsvector('portuguese', NEW.group_name), 'A') ||
            setweight(to_tsvector('portuguese', NEW.description), 'B')
        );
    ELSIF TG_OP = 'UPDATE' THEN
        IF NEW.group_name <> OLD.group_name OR NEW.description <> OLD.description THEN
            NEW.tsvectors := (
                setweight(to_tsvector('portuguese', NEW.group_name), 'A') ||
                setweight(to_tsvector('portuguese', NEW.description), 'B')
            );
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER group_search_update
BEFORE INSERT OR UPDATE ON group_
FOR EACH ROW
EXECUTE FUNCTION group_search_update();

CREATE INDEX idx_group_description ON group_ USING GIN (tsvectors);

-- Triggers

-- TRIGGER01: Enforces that only approved friends can view private profiles (BR01, BR07)
CREATE OR REPLACE FUNCTION enforce_profile_visibility_update()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.is_public = FALSE THEN
        IF NOT EXISTS (
            SELECT 1 FROM friendship
            WHERE (user_id1 = NEW.id AND user_id2 = current_user) OR
                  (user_id2 = NEW.id AND user_id1 = current_user)
        ) AND NOT EXISTS (
            SELECT 1 FROM administrators
            WHERE user_id = current_user
        ) THEN
            RAISE EXCEPTION 'Perfil privado. Acesso negado.';
        END IF;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_profile_visibility
BEFORE UPDATE ON users
FOR EACH ROW
EXECUTE FUNCTION enforce_profile_visibility_update();

-- TRIGGER02: Ensures users cannot send duplicate friend requests (BR02)
CREATE OR REPLACE FUNCTION enforce_friend_request_limit()
RETURNS TRIGGER AS $$
BEGIN
    IF EXISTS (
        SELECT 1 FROM friend_request
        WHERE sender_id = NEW.sender_id
```

```
          AND receiver_id = NEW.receiver_id
          AND request_status NOT IN ('denied')
    ) THEN
        RAISE EXCEPTION 'Not successful: cannot send more than one friend request to the same user.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_friend_request_limit
BEFORE INSERT ON friend_request
FOR EACH ROW
EXECUTE FUNCTION enforce_friend_request_limit();


-- TRIGGER03: Prevents multiple reactions from the same user on a single post (BR04)
CREATE OR REPLACE FUNCTION enforce_reaction_limit()
RETURNS TRIGGER AS $$
BEGIN
    -- Verificar se já existe uma reação do mesmo usuário para o mesmo target (post ou comentário)
    IF EXISTS (
        SELECT 1 FROM reaction
        WHERE target_id = NEW.target_id
          AND target_type = NEW.target_type
          AND user_id = NEW.user_id
    ) THEN
        RAISE EXCEPTION 'User already reacted to this target.';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_reaction_limit
BEFORE INSERT ON reaction
FOR EACH ROW
EXECUTE FUNCTION enforce_reaction_limit();



-- TRIGGER04: Ensures each post has content (text or media) (BR08)
CREATE OR REPLACE FUNCTION enforce_post_content()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.content IS NULL AND NEW.image1 IS NULL AND NEW.image2 IS NULL AND NEW.image3 IS NULL THEN
        RAISE EXCEPTION 'A post must contain text or at least one image.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_post_content
BEFORE INSERT OR UPDATE ON post
FOR EACH ROW
EXECUTE FUNCTION enforce_post_content();


-- TRIGGER05: Anonymizes user data upon account deletion, retaining content (BR05)
CREATE OR REPLACE FUNCTION anonymize_user_data()
RETURNS TRIGGER AS $$
BEGIN
    UPDATE comment_ SET user_id = 0 WHERE user_id = OLD.id;
    UPDATE reaction SET user_id = 0 WHERE user_id = OLD.id;
    UPDATE post SET user_id = 0 WHERE user_id = OLD.id;
    RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER trg_anonymize_user_data
AFTER DELETE ON users
FOR EACH ROW
EXECUTE FUNCTION anonymize_user_data();


-- TRIGGER06: Ensures users can only post in groups they belong to (BR11)
CREATE OR REPLACE FUNCTION enforce_group_posting()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.group_id IS NOT NULL THEN
        IF NOT EXISTS (
            SELECT 1 FROM group_member
            WHERE group_id = NEW.group_id AND user_id = NEW.user_id
        ) THEN
            RAISE EXCEPTION 'User must be a member of the group to post.';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_group_posting
BEFORE INSERT ON post
FOR EACH ROW
EXECUTE FUNCTION enforce_group_posting();



-- TRIGGER07: Requires group owner's approval for joining private groups (BR03)
CREATE OR REPLACE FUNCTION enforce_group_membership_control()
RETURNS TRIGGER AS $$
BEGIN
    IF (SELECT is_public FROM group_ WHERE id = NEW.group_id) = FALSE THEN
        IF NEW.request_status = 'Pending' THEN
            RAISE EXCEPTION 'Group membership requires owner approval.';
        END IF;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_enforce_group_membership_control
BEFORE INSERT OR UPDATE ON join_group_request
FOR EACH ROW
EXECUTE FUNCTION enforce_group_membership_control();


-- TRIGGER08: Validation of the reaction target (post or comment)
CREATE OR REPLACE FUNCTION validate_target_id()
RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.target_type = 'post') THEN
        IF NOT EXISTS (SELECT 1 FROM post WHERE id = NEW.target_id) THEN
            RAISE EXCEPTION 'Target ID % not found in post table.', NEW.target_id;
        END IF;
    ELSIF (NEW.target_type = 'comment') THEN
        IF NOT EXISTS (SELECT 1 FROM comment_ WHERE id = NEW.target_id) THEN
            RAISE EXCEPTION 'Target ID % not found in comment table.', NEW.target_id;
        END IF;
    ELSE
        RAISE EXCEPTION 'Invalid target type: %', NEW.target_type;
    END IF;

    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_target_id
BEFORE INSERT OR UPDATE ON reaction
FOR EACH ROW
EXECUTE FUNCTION validate_target_id();



-- Transactions
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;


-- Tran01
CREATE OR REPLACE FUNCTION inserir_post(content TEXT, visibility BOOLEAN)
RETURNS VOID AS $$
BEGIN
    INSERT INTO post (content, is_public, user_id)
    VALUES ($1, $2, COALESCE(current_user_id, 1));
END;
$$ LANGUAGE plpgsql;

-- Tran02
CREATE OR REPLACE FUNCTION add_comment(
    postId INT,
    userId INT,
    commentContent TEXT
) RETURNS VOID AS $$
DECLARE
    postOwnerId INT;
    commentId INT;
BEGIN
    SELECT user_id INTO postOwnerId
    FROM post
    WHERE post_id = postId;

    IF postOwnerId IS NULL THEN
        RAISE EXCEPTION 'No valid post found.';
    END IF;

    INSERT INTO comment_ (post_id, user_id, comment_content, commentDate)
    VALUES (postId, userId, commentContent, NOW())
    RETURNING id INTO commentId;

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || userId || ' commented on your post.', FALSE, NOW(), postOwnerId);

    INSERT INTO comment_notification (notification_id, comment_id)
    VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), commentId);
END;
$$ LANGUAGE plpgsql;



--Tran03
CREATE OR REPLACE FUNCTION add_reaction(postId INT, userId INT, reactionType TEXT)
RETURNS VOID AS $$
DECLARE
    postOwnerId INT;
    reactionId INT;
BEGIN
    SELECT user_id INTO postOwnerId
    FROM post
    WHERE post_id = postId;

    INSERT INTO reaction (reactionType, reaction_date, post_id, user_id)
    VALUES (reactionType, NOW(), postId, userId)
    RETURNING reaction_id INTO reactionId;
```

```
    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || userId || ' reacted to your post with ' || reactionType, FALSE, NOW(), postOwnerId);

    INSERT INTO reaction_notification (notification_id, reaction_id)
    VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), reactionId);
END; $$ LANGUAGE plpgsql;

--Tran04
CREATE OR REPLACE FUNCTION update_user_info(userId INT, newName TEXT, newEmail TEXT, newProfilePicture TEXT)
RETURNS VOID AS $$
BEGIN
    UPDATE users
    SET username = newName, email = newEmail, profile_picture = newProfilePicture
    WHERE id = userId;
END; $$ LANGUAGE plpgsql;


-- Tran05
CREATE OR REPLACE FUNCTION delete_post(postId INT)
RETURNS VOID AS $$
BEGIN
    DELETE FROM comment_ WHERE post_id = postId;
    DELETE FROM reaction WHERE post_id = postId;
    DELETE FROM post WHERE post_id = postId;
END;
$$ LANGUAGE plpgsql;


-- Tran06
CREATE OR REPLACE FUNCTION send_friend_request(sender_id INT, receiver_id INT)
RETURNS VOID AS $$
DECLARE
    notification_id INT;
BEGIN
    INSERT INTO friend_request (request_date, request_status, sender_id, receiver_id)
    VALUES (NOW(), 'pending', sender_id, receiver_id);

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || sender_id || ' sent you a friend request.', FALSE, NOW(), receiver_id);

    notification_id := currval(pg_get_serial_sequence('notification', 'notification_id'));

    INSERT INTO friend_request_notification (notification_id, friend_request_id)
    VALUES (notification_id, (SELECT MAX(request_id) FROM friend_request WHERE sender_id = sender_id AND receiver_id = r
END;
$$ LANGUAGE plpgsql;


-- Tran07
CREATE OR REPLACE FUNCTION accept_friend_request(request_id INT, user_id1 INT, user_id2 INT)
RETURNS VOID AS $$
BEGIN
    UPDATE friend_request
    SET request_status = 'accepted'
    WHERE id = request_id;

    INSERT INTO friendship (userId1, userId2)
    VALUES (user_id1, user_id2);
END;
$$ LANGUAGE plpgsql;

-- Tran08
CREATE OR REPLACE FUNCTION create_group(group_name TEXT, description TEXT, is_public BOOLEAN, owner_id INT)
RETURNS VOID AS $$
BEGIN
    INSERT INTO "group" (groupName, description, is_public, ownerId)
```

```
    VALUES (group_name, description, is_public, owner_id);
END;
$$ LANGUAGE plpgsql;


-- Tran09
CREATE OR REPLACE FUNCTION delete_user(user_id INT)
RETURNS VOID AS $$
BEGIN
    UPDATE comment
    SET userId = NULL,
        content = CONCAT('Deleted User: ', content)
    WHERE userId = user_id;

    UPDATE post
    SET userId = NULL,
        content = CONCAT('Deleted User Post: ', content)
    WHERE userId = user_id;

    UPDATE saved_post
    SET userId = NULL
    WHERE userId = user_id;

    UPDATE friend_request
    SET senderId = NULL
    WHERE senderId = user_id;

    UPDATE friend_request
    SET receiverId = NULL
    WHERE receiverId = user_id;

    UPDATE group_member
    SET user_id = NULL
    WHERE user_id = user_id;

    UPDATE group_owner
    SET user_id = NULL
    WHERE user_id = user_id;

    DELETE FROM users
    WHERE id = user_id;
END;
$$ LANGUAGE plpgsql;


-- Tran10
CREATE OR REPLACE FUNCTION save_post(user_id INT, post_id INT)
RETURNS VOID AS $$
BEGIN
    INSERT INTO saved_post (user_id, postId)
    VALUES (user_id, post_id);
END;
$$ LANGUAGE plpgsql;


-- Tran11
CREATE OR REPLACE FUNCTION remove_saved_post(user_id INT, post_id INT)
RETURNS VOID AS $$
BEGIN
    DELETE FROM saved_post
    WHERE user_id = user_id AND postId = post_id;
END;
$$ LANGUAGE plpgsql;


-- Tran12
CREATE OR REPLACE FUNCTION request_to_join_group(group_id INT, user_id INT)
```

```
RETURNS VOID AS $$
DECLARE
    groupOwnerId INT;
    notification_id INT;
BEGIN
    INSERT INTO join_group_request (group_id, user_id, request_status, requested_at)
    VALUES (group_id, user_id, 'pending', NOW());

    SELECT owner_id INTO groupOwnerId
    FROM group_
    WHERE group_id = group_id;

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || user_id || ' has requested to join your group.', FALSE, NOW(), groupOwnerId);

    notification_id := LASTVAL();

    INSERT INTO group_request_notification (notification_id, group_request_id)
    VALUES (notification_id, (SELECT MAX(request_id) FROM join_group_request WHERE group_id = group_id AND user_id = us
END;
$$ LANGUAGE plpgsql;



-- Tran13
CREATE OR REPLACE FUNCTION add_post(
    p_user_id INT,
    p_group_id INT,
    p_content TEXT
)
RETURNS VOID AS $$
DECLARE
    groupOwnerId INT;
    groupMemberIds INT[];
    new_post_id INT;
    member_id INT;
BEGIN
    INSERT INTO post (user_id, group_id, content, post_date)
    VALUES (p_user_id, p_group_id, p_content, NOW())
    RETURNING post_id INTO new_post_id;

    SELECT owner_id INTO groupOwnerId
    FROM group_
    WHERE group_id = p_group_id;

    SELECT ARRAY_AGG(user_id) INTO groupMemberIds
    FROM group_member
    WHERE group_id = p_group_id;

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || p_user_id || ' has posted in your group.', FALSE, NOW(), groupOwnerId);

    FOREACH member_id IN ARRAY groupMemberIds LOOP
        INSERT INTO notification (content, is_read, notification_date, user_id)
        VALUES ('User ' || p_user_id || ' has posted in the group.', FALSE, NOW(), member_id);

        INSERT INTO group_post_notification (notification_id, post_id)
        VALUES (currval(pg_get_serial_sequence('notification', 'notification_id')), new_post_id);
    END LOOP;
END $$ LANGUAGE plpgsql;



-- Tran14
CREATE OR REPLACE FUNCTION accept_join_group_request(
    p_request_id INT  -- Declare request_id as a parameter
)
RETURNS VOID AS $$
```

```
DECLARE
    requesterId INT;
    groupId INT;
    groupOwnerId INT;
    notification_id INT;
BEGIN
    -- Select user_id and group_id from the join_group_request table
    SELECT user_id, group_id INTO requesterId, groupId
    FROM join_group_request
    WHERE request_id = p_request_id;  -- Use the parameter instead

    -- Update the request status to accepted
    UPDATE join_group_request
    SET request_status = 'accepted'
    WHERE request_id = p_request_id;  -- Use the parameter instead

    -- Insert the new group member
    INSERT INTO group_member (user_id, group_id)
    VALUES (requesterId, groupId);

    -- Get the group owner's ID
    SELECT owner_id INTO groupOwnerId
    FROM group_
    WHERE group_id = groupId;

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('Your request to join the group has been accepted.', FALSE, NOW(), requesterId);

    INSERT INTO notification (content, is_read, notification_date, user_id)
    VALUES ('User ' || requesterId || ' has joined your group.', FALSE, NOW(), groupOwnerId);

    notification_id := currval(pg_get_serial_sequence('notification', 'notification_id'));


    INSERT INTO group_request_notification (notification_id, request_id)
    VALUES (notification_id, p_request_id);
END $$ LANGUAGE plpgsql;
```

A.2. Database population

```
INSERT INTO users (username, email, profile_picture, password, is_public)
VALUES
    ('alice_wonder', 'alice@example.com', 'alice.jpg', '$2y$10$rX7CLGWOUaeAKP6ACma35.e9bVB5QqD5hLlUrU.nhxgdI2qWd9v7W', 
    ('bob_builder', 'bob@example.com', 'bob.jpg', '$2y$10$0xP8NZro/7udYYA0IA8Zhey919ccCDwUjSsj7ulYJlXpUXsSJ306G', TRUE)
    ('charlie_chaplin', 'charlie@example.com', 'charlie.jpg', 'securepassword3', FALSE),
    ('daisy_duck', 'daisy@example.com', 'daisy.jpg', 'securepassword4', TRUE),
    ('edgar_allan', 'edgar@example.com', 'edgar.jpg', 'securepassword5', TRUE),
    ('fiona_fairy', 'fiona@example.com', 'fiona.jpg', 'securepassword6', FALSE),
    ('george_gremlin', 'george@example.com', 'george.jpg', 'securepassword7', TRUE),
    ('hannah_hacker', 'hannah@example.com', 'hannah.jpg', 'securepassword8', TRUE),
    ('ian_icecream', 'ian@example.com', 'ian.jpg', 'securepassword9', TRUE),
```

Revision history

Changes made to the first submission:

- 20/11/2024: Added highlights in sql code, schema creation and increased the growth of some relations.
- 25/11/2024: Changed user_ to users in the table name of the users; changed the user_password field to password in the users table; updated trigger1 to add permissions to administrators to see private profiles; updated some of the populate users password to be hashed.
- 2/12/2024: Changed reaction table to handle post and comments reactions, added trigger 8 to validate the target type of reaction, updated trigger 3, updated UML.
- 8/12/2024: Fixed creation of notification table, column names were wrong.
- 22/12/2024: Deleted Trigger 1.
- 23/12/2024: Change in the TRIGGER04: user_id = null -> user_id = 0.

GROUP2453, 14/10/2024

- Group member 1 Maria Vieira[editor], up202204802@g.uporto.pt
- Group member 2 Marta Cruz, up202205028@g.uporto.pt
- Group member 3 Bruna Scafutto, up202402704@g.uporto.pt
- Group member 4 Duarte Marques, up202204973@g.uporto.pt