ebd

Last edited by **Rui Pedro da Silva Cruz** 3 months ago

EBD: Database Specification Component

Planora is a flexible and efficient project management platform that empowers individuals and teams to organize, track, and complete their projects with ease. Designed with user-friendly navigation and a responsive interface, it enables seamless collaboration and task management across all devices.

A4: Conceptual Data Model

The objective of this artifact is to illustrate the organizational entities in the Planora system, their relationships, and associated attributes, serving as a high-level blueprint for database structure and integrity within the project management platform.

1. Class diagram

The UML class diagram corresponding to Figure 1, shown below, contains the identification of the organisational entities, the relationships established between them, along with their respective attributes and domains, and the multiplicity between these relationships in the Planora system.

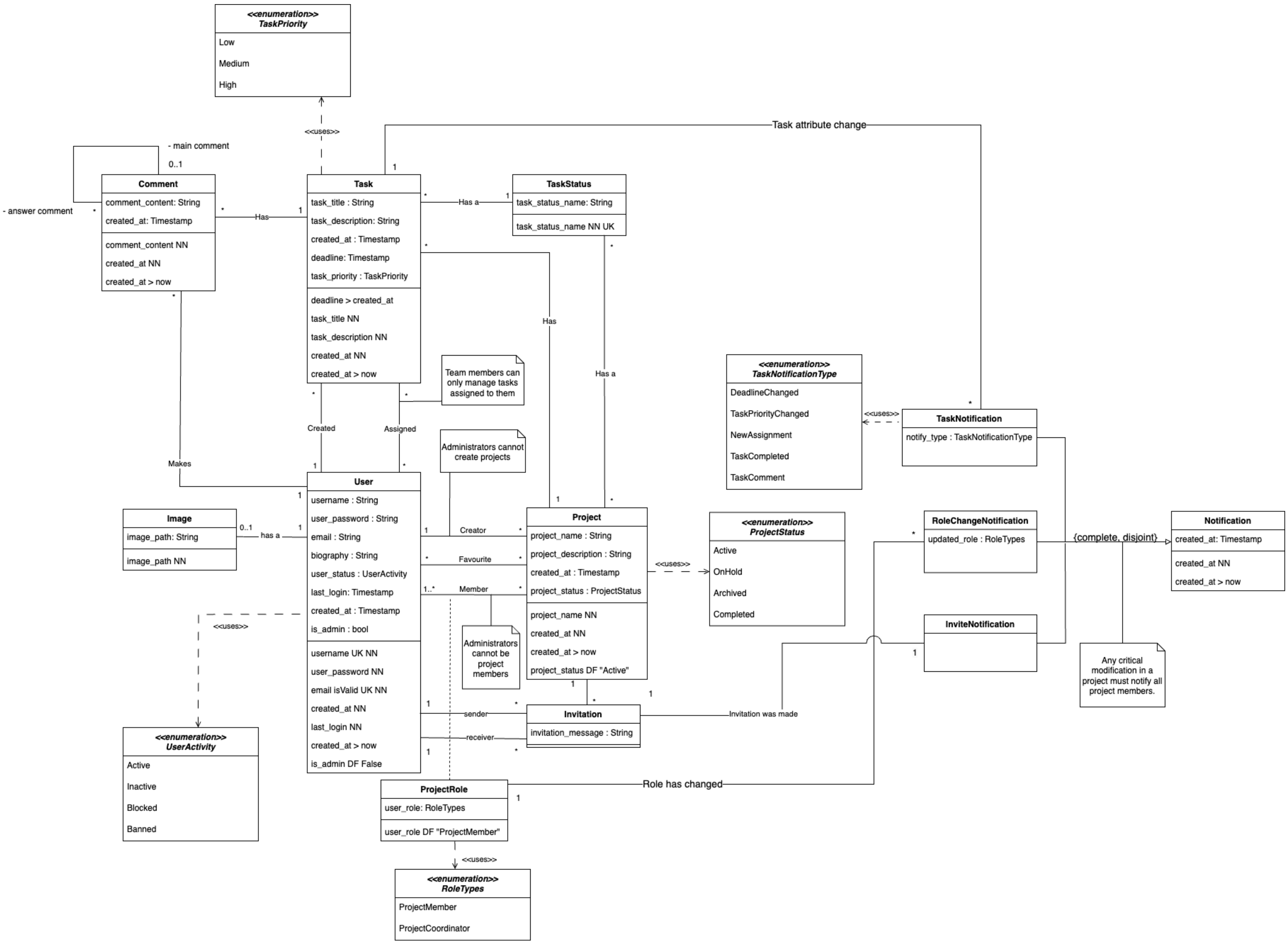

Figure 1: Planora conceptual data model in UML.

2. Additional Business Rules

In this section, we concluded that no additional business rules were identified beyond those already captured in the UML diagram.

A5: Relational Schema, validation and schema refinement

This artifact presents the relational schema derived from the conceptual data model, detailing attributes, keys, integrity constraints, and domains for each entity. It includes validation through functional dependencies, ensuring data consistency and normalization to BCNF, with schema refinement applied as needed to create a robust, redundancy-free database structure.

1. Relational Schema

This section provides the relational schemas derived from the conceptual data model analysis, detailing each entity's attributes, domains, primary keys, foreign keys, and essential integrity constraints, including UNIQUE, DEFAULT, CHECK and NOT NULL specifications.

Relation reference	Relation Compact Notation
R01	user (user_id, username **NN**, user_password **NN**, email **UK** **NN**, biography, user_status **DF** Active **CK** user_status **IN** UserActivity, last_login **NN**, created_at **NN**, is_admin **DF** False)
R02	project (project_id, project_name **NN**, project_description, created_at **NN**, project_status **DF** Active **CK** project_status **IN** ProjectStatus, creator_id → user)
R03	invitation (invitation_id, invitation_message, sender_id **NN** → user, receiver_id **NN** → user, project_id → project)
R04	task (task_id, task_title **NN**, task_description **NN**, created_at **NN**, deadline, task_priority **CK** task_priority **IN** TaskPriority, project_id → project, user_id → user, task_status_id → taskstatus)
R05	assigned_task (user_id → user, task_id → task)
R06	comment (comment_id, comment_content **NN**, created_at **NN**, task_id → task)
R07	comment_relation (child_id → comment, parent_id → comment)
R08	projectrole (user_id → user, project_id → project, user_role **DF** ProjectMember **CK** user_role **IN** RoleTypes)
R09	favourite_project (user_id → user, project_id → project)
R10	rolechangenotification (role_change_notification_id, created_at **NN**, updated_role **NN** **CK** updated_role **IN** RoleTypes, updated_user **NN** → projectrole, updated_project **NN** → projectrole)
R11	invitenotification (invite_notification_id, created_at **NN**, user_id **NN** → user)
R12	tasknotification (task_notification_id, created_at **NN**, task_id **NN** → task, notify_type **NN** **CK** notify_type **IN** TaskNotificationType)
R13	image (image_id, image_path **NN**, user_id **UK** → user)
R14	taskstatus (task_status_id, task_status_name)
R15	project_task_status (task_status_id → taskstatus, project_id → project)

The object-oriented style in this UML schema offers significant advantages. By using inheritance, the `Notification` class provides common attributes to its specific subtypes (`TaskNotification`, `RoleChangeNotification`, and `InviteNotification`), eliminating redundancy. The `{complete, disjoint}` constraint ensures that each notification instance is categorised into one specific subtype, meaning the generalisation is exhaustive. This completeness removes the need for an additional general `Notification` table, simplifying data handling and ensuring clarity.

2. Domains

This section defines the attribute domains used in the schema.

Domain Name	Domain Specification
UserActivity	ENUM ('Active', 'Inactive', 'Blocked', 'Banned')
ProjectStatus	ENUM ('Active', 'Completed', 'OnHold', 'Archived')
TaskPriority	ENUM ('Low', 'Medium', 'High')
RoleTypes	ENUM ('ProjectCoordinator', 'ProjectMember')
TaskNotificationType	ENUM ('DeadlineChanged', 'TaskPriorityChanged', 'NewAssignment', 'TaskCompleted', 'TaskComment')

3. Schema validation

This section validates the schema by analyzing functional dependencies and ensuring normalization to BCNF for all tables.

TABLE R01	user
Keys	{ user_id }, { email }

TABLE R01	user
Functional Dependencies:	
FD0101	user_id → { username, user_password, email, biography, user_status, last_login, created_at, is_admin }
FD0102	email → { user_id, username, user_password, biography, user_status, last_login, created_at, is_admin }
NORMAL FORM	BCNF

TABLE R02	project
Keys	{ project_id }
Functional Dependencies:	
FD0201	project_id → { project_name, project_description, created_at, project_status, creator_id }
NORMAL FORM	BCNF

TABLE R03	invitation
Keys	{ invitation_id }
Functional Dependencies:	
FD0301	invitation_id → { invitation_message, sender_id, receiver_id, project_id }
NORMAL FORM	BCNF

TABLE R04	task
Keys	{ task_id }
Functional Dependencies:	
FD0401	task_id → { task_title, task_description, created_at, deadline, task_status_id, task_priority, project_id, user_id}
NORMAL FORM	BCNF

TABLE R05	assigned_task
Keys	{ user_id, task_id }
Functional Dependencies:	
FD0501	{ user_id, task_id } → { }
NORMAL FORM	BCNF

TABLE R06	comment
Keys	{ comment_id }
Functional Dependencies:	
FD0601	comment_id → { comment_content, created_at, task_id }
NORMAL FORM	BCNF

TABLE R07	comment_relation
Keys	{ child_id }
Functional Dependencies:	
FD0701	child_id → { parent_id }

TABLE R07	comment_relation
NORMAL FORM	BCNF

TABLE R08	projectrole
Keys	{ user_id, project_id }
Functional Dependencies:	
FD0801	{ user_id, project_id } → { user_role }
NORMAL FORM	BCNF

TABLE R09	favourite_project
Keys	{ user_id, project_id }
Functional Dependencies:	
FD0901	{ user_id, project_id } → { }
NORMAL FORM	BCNF

TABLE R10	rolechangenotification
Keys	{ role_change_notification_id }
Functional Dependencies:	
FD1001	role_change_notification_id → { created_date, updated_role, updated_user, updated_project }
NORMAL FORM	BCNF

TABLE R11	invitenotification
Keys	{ invite_notification_id }
Functional Dependencies:	
FD1101	invite_notification_id → { created_at, user_id }
NORMAL FORM	BCNF

TABLE R12	tasknotification
Keys	{ task_notification_id }
Functional Dependencies:	
FD1201	task_notification_id → { created_at , task_id, notify_type }
NORMAL FORM	BCNF

TABLE R13	image
Keys	{ image_id }
Functional Dependencies:	
FD1301	image_id → { image_path, user_id }
NORMAL FORM	BCNF

TABLE R14	taskstatus
Keys	{ task_status_id }
Functional Dependencies:	
FD1401	task_status_id → { task_status_name }
NORMAL FORM	BCNF

TABLE R15	project_task_status
Keys	{ task_status_name, project_id }
Functional Dependencies:	
FD1501	{ task_status_name, project_id } → { }
NORMAL FORM	BCNF

After analysing the functional dependencies and keys of each relationship, we concluded that the schema was already standardised in BCNF and that no changes were necessary. This is due to the fact that in every relationship there is a primary key, whether single or composite, which determines all the other non-key attributes, ensuring that the determinant is always a superkey. We further verified that all functional dependencies align with BCNF principles, eliminating any partial and transitive dependencies and confirming that no additional decomposition is required. Thus, the schema fully complies with BCNF requirements, ensuring a robust design free from redundancy and undue dependencies.

A6: Indexes, triggers, transactions and database population

This artifact provides the PostgreSQL code for the physical database schema and data population, implementing data integrity through triggers, defining indexes, and incorporating user-defined functions. It includes essential transactions to maintain data accuracy during database interactions, along with an explanation of isolation levels to ensure consistency and prevent conflicts.

1. Database Workload

A study of the predicted system load (database load).

Relation reference	Relation Name	Order of magnitude	Estimated growth
R01	user	10k(tens of thousands)	hundreds per day
R02	project	10k(tens of thousands)	tens per day
R03	invitation	10k(tens of thousands)	tens per day
R04	task	100k(hundreds of thousands)	hundreds per day
R05	assigned_task	100k(hundreds of thousands)	hundreds per day
R06	comment	100k(hundreds of thousands)	hundreds per day
R07	comment_relation	100k(hundreds of thousands)	hundreds per day
R08	projectrole	100k(hundreds of thousands)	hundreds per day
R09	favourite_project	1k(thousands)	tens per day
R10	rolechangenotification	10k(hundreds of thousands)	hundreds per day
R11	invitenotification	10k(hundreds of thousands)	hundreds per day
R12	tasknotification	10k(hundreds of thousands)	hundreds per day
R13	image	10k(tens of thousands)	hundreds per day
R14	taskstatus	100(hundreds)	units per day
R15	project_task_status	100(hundreds)	units per day

2. Proposed Indexes

This section defines the indexes used in the database to optimize performance and enable full-text search capabilities, enhancing the efficiency of query execution for common filters and search operations.

2.1. Performance Indexes

Performance indexes are designed to improve retrieval speed by indexing frequently filtered attributes. B-tree indexes support efficient data access and sorting, with clustering applied where it benefits retrieval performance.

Index	**IDX01**
Relation	task
Attribute	deadline
Type	B-tree
Clustering	Yes
Cardinality	Medium
Justification	The `task` table is frequently filtered by each task deadline. A B-tree index is the most appropriate type of index, since this filtering is done using comparisons rather than exact matches and it provides efficient sorting. In addition, clustering is beneficial as tasks will be accessed in deadline order.

`SQL code`	`CREATE INDEX idx_task_deadline ON Task USING btree (deadline);`
	`CLUSTER Task USING idx_task_deadline;`

Index	**IDX02**
Relation	task
Attribute	task_status_id
Type	B-tree
Clustering	No
Cardinality	Low
Justification	The `task` table is frequently filtered by each task status(e.g., pending, completed, in progress). A B-tree index is the most appropriate type of index, since this filtering is often done using comparisons rather than exact matches. In addition, clustering is not necessary as tasks will be queried in varying orders such as `deadline` or `priority`.
`SQL code`	`CREATE INDEX idx_task_status ON Task USING btree (task_status_id);`

Index	**IDX03**
Relation	project
Attribute	project_status

Type	B-tree
Clustering	Yes
Cardinality	Low
Justification	The `project` table is frequently queried to display projects by `project_status`. A B-tree index on this attribute allows efficient filtering, especially when displaying grouped project statuses. Clustering is beneficial here as it allows related rows to be stored close together for quicker retrieval.

`SQL code`	`CREATE INDEX idx_project_status ON Project USING btree (project_status);`
	`CLUSTER Project USING idx_project_status;`

2.2. Full-text Search Indexes

To fulfill the project's specifications for full-text search capabilities, we have developed Full-Text Search (FTS) indexes. These indexes are strategically implemented on the tables and attributes we anticipate will be queried most frequently, thereby enhancing text search performance. Detailed information about these indexes can be found in the following tables:

Index	**IDX11**
Relation	task
Attribute	task_title, task_description
Type	GIN
Clustering	No

Justification	In order to provide full-text search features to look for a task by its title and description. GIN is the most appropriate index type as these attributes are not expected to change frequently.

`SQL code`

```
ALTER TABLE Task
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION task_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.task_title), 'A') ||
         setweight(to_tsvector('english', NEW.task_description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.task_title <> OLD.task_title OR NEW.task_description <> OLD.task_description) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.task_title), 'A') ||
             setweight(to_tsvector('english', NEW.task_description), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER task_search_update
 BEFORE INSERT OR UPDATE ON Task
 FOR EACH ROW
 EXECUTE PROCEDURE task_search_update();

CREATE INDEX task_search_idx ON Task USING GIN (tsvectors);
```

Index	**IDX12**
Relation	project
Attribute	project_name, project_description
Type	GIN
Clustering	No

Justification	In order to provide full-text search features to look for a project by its name and description. GIN is the most appropriate index type a attributes are not expected to change frequently.

`SQL code`

```
ALTER TABLE Project
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION project_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.project_name), 'A') ||
         setweight(to_tsvector('english', NEW.project_description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.project_name <> OLD.project_name OR NEW.project_description <> OLD.project_description
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.project_name), 'A') ||
             setweight(to_tsvector('english', NEW.project_description), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER project_search_update
 BEFORE INSERT OR UPDATE ON Project
 FOR EACH ROW
 EXECUTE PROCEDURE project_search_update();

CREATE INDEX project_search_idx ON Project USING GIN (tsvectors);
```

3. Triggers

To implement complex integrity rules, we've defined the required triggers, detailing the activating events, conditions, and corresponding actions. Triggers are also employed to maintain up-to-date full-text indexes, ensuring accurate and current search capabilities.

Trigger	**TRIGGER01 - Updating Completed Tasks**

Description

Activated when the status of a task is changed to ‘Completed’. If the status has indeed changed to ‘Completed’, the system sends noti to the project members and the person responsible for the task informing them of the completion.

It is in line with **BR03**, which states that team members can see and manage their own tasks, and **BR02**, which makes coordinators res for monitoring the progress of tasks.

`SQL code`

```
CREATE OR REPLACE FUNCTION notify_task_completion() RETURNS TRIGGER AS $$
DECLARE
    completed_status_id INT;
BEGIN
    SELECT task_status_id INTO completed_status_id
    FROM TaskStatus
    WHERE task_status_name = 'Completed';

    IF NEW.task_status_id = completed_status_id AND OLD.task_status_id IS DISTINCT FROM NEW.task_status_
        INSERT INTO TaskNotification (created_at, task_id, notify_type)
        VALUES (NOW(), NEW.task_id, 'TaskCompleted');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_completion_trigger
AFTER UPDATE OF task_status_id ON Task
FOR EACH ROW
EXECUTE FUNCTION notify_task_completion();
```

Trigger	TRIGGER02 - Appointment of Project Coordinator

Description

Triggered when the coordinator of a project is changed. If a new coordinator is appointed, a notification is sent to all project members informing them of the new leadership.

It relates to **BR04**, which defines the hierarchical permissions of users, and **BR02**, which makes coordinators responsible for effective team management.

`SQL code`

```
CREATE OR REPLACE FUNCTION notify_coordinator_change() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO RoleChangeNotification (created_at, updated_user, updated_project, updated_role)
    VALUES (NOW(), NEW.user_id, NEW.project_id, NEW.user_role);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER coordinator_change_trigger
AFTER UPDATE OF user_role ON ProjectRole
FOR EACH ROW
WHEN (NEW.user_role = 'ProjectCoordinator')
EXECUTE FUNCTION notify_coordinator_change();
```

Trigger	TRIGGER03 - Assigning Tasks

Description	This occurs when a user is assigned a new task. If the task has been successfully assigned, the system sends a notification to the user to inform them of their new responsibility. It supports **BR03**, which defines that members can only see and manage their own tasks, and **BR02**, which delegates responsibility for distributing tasks to coordinators.

`SQL code`

```sql
CREATE OR REPLACE FUNCTION notify_task_assignment() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'NewAssignment');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_assignment_trigger
AFTER INSERT ON AssignedTask
FOR EACH ROW
EXECUTE FUNCTION notify_task_assignment();
```

Trigger	TRIGGER04 - Project invitations
Description	Activated when an invitation is created for a user to participate in a project. If the invitation is issued correctly, a notification will alert the user to either accept or decline that invitation. It relates to **BR04**, which defines the organisation of users into groups with appropriate permissions, ensuring that the invitation process respects this hierarchy.

`SQL code`

```sql
CREATE OR REPLACE FUNCTION notify_invitation_creation() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO InviteNotification (created_at, user_id)
    VALUES (NOW(), NEW.receiver_id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER invitation_creation_trigger
AFTER INSERT ON Invitation
FOR EACH ROW
EXECUTE FUNCTION notify_invitation_creation();
```

Trigger	TRIGGER05 - Comments on Tasks

<table>
<tr><td><b>Description</b></td><td>Triggered when a comment is added to a task. If the comment has been correctly associated with the task, a notification is sent to the project members and the person responsible for the task, informing them of the new comment.

It supports <b>BR03</b>, which allows members to interact in the tasks assigned to them, facilitating collaboration and communication within the team.</td></tr>
<tr><td><code>SQL code</code></td><td><pre>
CREATE OR REPLACE FUNCTION notify_task_comment() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'TaskComment');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_comment_trigger
AFTER INSERT ON "Comment"
FOR EACH ROW
EXECUTE FUNCTION notify_task_comment();
</pre></td></tr>
</table>

<table>
<tr><th>Trigger</th><th>TRIGGER06 - Deadline Changes in Tasks</th></tr>
<tr><td><b>Description</b></td><td>Triggered when the deadline of a task is changed. All the project members assigned with that task will be notified about that change.

It is in line with <b>BR03</b>, which allows members to interact only on the tasks assigned to them, and <b>BR06</b>, which ensures that entry and exit dates are valid and consistent.</td></tr>
<tr><td><code>SQL code</code></td><td><pre>
CREATE OR REPLACE FUNCTION notify_deadline_change() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'DeadlineChanged');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER deadline_change_trigger
AFTER UPDATE OF deadline ON Task
FOR EACH ROW
WHEN (OLD.deadline IS DISTINCT FROM NEW.deadline)
EXECUTE FUNCTION notify_deadline_change();
</pre></td></tr>
</table>

4. Transactions

The following transactions are employed to ensure data integrity when multiple operations are necessary. They are exemples of possible transactions, with created data. They guarantee that all required actions are executed successfully, maintaining the consistency of the database.

Transaction	TRAN01
Description	Creation of a new project
Justification	When a user creates a project, they are automatically assigned as the project coordinator, and the project is created with an "Active" status. This involves inserting data into both the `Project` and `ProjectRole` tables, and these two actions need to be grouped in a single transaction to maintain data consistency.

Isolation level	`Serializable` is chosen to prevent "phantom reads," where new records in related tables could unexpectedly affect the transaction.

`SQL code`

```
DECLARE
    new_project_id INT;
    creator_id INT := ?;
BEGIN
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    INSERT INTO Project (project_name, project_status, project_description, created_at, creator_id)
    VALUES ($project_name, 'Active', $project_description, NOW(), creator_id)
    RETURNING project_id INTO new_project_id;

    INSERT INTO ProjectRole (user_id, project_id, user_role)
    VALUES ($user_id, new_project_id, 'ProjectCoordinator');

EXCEPTION
    WHEN OTHERS THEN
        RAISE;
END;
```

Transaction	**TRAN02**
Description	Create a comment reply
Justification	When a reply is added to a comment, two tables are involved: `Comment`, where the reply content is saved, and `CommentRelation`, which tracks the parent-child relationships between comments. Using a transaction ensures that both the comment and its relationship to the original comment are stored consistently
Isolation level	Read Commited is sufficient since the risk of interference from other transactions is low.

`SQL code`

```
DECLARE
    created_comment_id INT;
BEGIN
    SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    INSERT INTO "Comment" (comment_content, created_at, task_id)
    VALUES ($comment_content, NOW(), $task_id);
    RETURNING comment_id INTO created_comment_id;


    INSERT INTO CommentRelation (child_id, parent_id)
    VALUES (created_comment_id, $parent_id);
    END IF;
END;
```

Transaction	**TRAN03**
Description	Creating a user
Justification	User creation involves storing information across the `User` and `Image` tables. First, the user data (username, password, email, etc.) is inserted into the `User` table, and then the profile image path is added to the `Image` table. We use transactions to prevent a scenario where a user is created without a profile image path or vice versa.

Isolation level	Serialized is isolated from other transactions, preventing issues like duplicate usernames or missing image paths.
`SQL code`	

```
DECLARE
    new_user_id INT;
BEGIN
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
    INSERT INTO "User" (username, user_password, email, biography, last_login, creation_date)
    VALUES ($username, $user_password, $email, $biography, NOW(), NOW())
    RETURNING user_id INTO new_user_id;

    INSERT INTO Image (image_path, user_id)
    VALUES ($image_path, new_user_id);
END;
```

Transaction	**TRAN04**
Description	Archived a project
Justification	Only project coordinators should have the authority to archive a project, so this transaction verifies the user's role before proceeding. Once confirmed, the project status is updated in the `Project` table.
Isolation level	Serialized is used to ensure that no other transactions can alter the project or user roles during this process. It prevents other project to be archived.
`SQL code`	

```
DECLARE
    is_coordinator BOOLEAN;
BEGIN
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    SELECT EXISTS (
        SELECT 1
        FROM ProjectRole
        WHERE user_id = $user_id AND project_id = $project_id AND user_role = 'ProjectCoordinator'
    ) INTO is_coordinator;

    IF NOT is_coordinator THEN
        RAISE EXCEPTION 'User is not the project coordinator.';
    END IF;

    UPDATE Project
    SET project_status = 'Archived'
    WHERE project_id = project_id;
END;
```

Transaction	**TRAN05**
Description	Deleting a user
Justification	When a user is deleted, their profile picture and other related data are also removed. The project is configured with cascading deletes, meaning that deleting a user will automatically delete all records associated with their user_id across relevant tables. This setup helps maintain data integrity by ensuring that no orphaned records remain.

Isolation level	Serializable isolation level is used to ensure that no other transactions can modify the user or user's image during this deletion process.

`SQL code`

```
BEGIN
    SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

    DELETE FROM "User"
    WHERE user_id = $user_id;

    DELETE FROM Image
    WHERE user_id = $user_id;
END;
```

Annex A. SQL Code

This annex includes the database scripts relevant to the EBD component. The scripts are presented as separate elements:

- Database Creation Script: This script contains all the code necessary to build and rebuild the database. It covers the creation of tables, relationships, constraints, and any other required database structures.
- Database Population Script: This script provides a set of sample data tuples suitable for testing. The data entries have plausible values to effectively test the fields and functionalities of the database.

The complete code for each script is available in the group's Git repository. Please refer to the repository to access the full scripts.

A.1. Database schema

In this section, we present the complete Planora database schema, including all SQL code necessary for creating the database. This comprehensive schema details the structure of the database—encompassing tables, relationships, constraints, indexes, and any additional features as specified by the project requirements.

```
DROP SCHEMA IF EXISTS lbaw24135 CASCADE;

CREATE SCHEMA IF NOT EXISTS lbaw24135;
SET search_path TO lbaw24135;

-- Domains

CREATE TYPE UserActivity AS ENUM ('Active', 'Inactive', 'Blocked', 'Banned');
CREATE TYPE ProjectStatus AS ENUM ('Active', 'Completed', 'Archived', 'OnHold');
CREATE TYPE TaskPriority AS ENUM ('Low', 'Medium', 'High');
CREATE TYPE RoleTypes AS ENUM ('ProjectCoordinator', 'ProjectMember');
CREATE TYPE TaskNotificationType AS ENUM ('DeadlineChanged', 'TaskPriorityChanged', 'NewAssignment', 'TaskCompleted', '

-- Tables

CREATE TABLE "User"(
    user_id SERIAL PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    user_password VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    biography TEXT,
    user_status UserActivity DEFAULT 'Active',
    last_login TIMESTAMP NOT NULL,
    created_at TIMESTAMP NOT NULL,
    is_admin BOOLEAN DEFAULT FALSE
);

CREATE TABLE TaskStatus (
        task_status_id SERIAL PRIMARY KEY,
    task_status_name VARCHAR(32)
);

CREATE TABLE Project(
    project_id SERIAL PRIMARY KEY,
    project_name VARCHAR(255) NOT NULL,
    project_description TEXT,
    created_at TIMESTAMP NOT NULL,
    project_status ProjectStatus DEFAULT 'Active',
    creator_id INT NOT NULL,
    FOREIGN KEY (creator_id) REFERENCES "User"(user_id)
);

CREATE TABLE Invitation(
    invitation_id SERIAL PRIMARY KEY,
    invitation_message TEXT,
    sender_id INT NOT NULL,
    receiver_id INT NOT NULL,
    project_id INT NOT NULL,
    FOREIGN KEY (sender_id) REFERENCES "User"(user_id),
    FOREIGN KEY (receiver_id) REFERENCES "User"(user_id),
    FOREIGN KEY (project_id) REFERENCES Project(project_id)
);

CREATE TABLE Task(
    task_id SERIAL PRIMARY KEY,
    task_title VARCHAR(255) NOT NULL,
    task_description TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    deadline TIMESTAMP,
    task_status_id INT,
    task_priority TaskPriority,
    project_id INT NOT NULL,
    user_id INT NOT NULL,
    FOREIGN KEY (task_status_id) REFERENCES TaskStatus(task_status_id),
    FOREIGN KEY (project_id) REFERENCES Project(project_id),
    FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE AssignedTask(
    user_id INT,
    task_id INT,
    PRIMARY KEY (user_id, task_id),
    FOREIGN KEY (user_id) REFERENCES "User"(user_id),
    FOREIGN KEY (task_id) REFERENCES Task(task_id)
);

CREATE TABLE "Comment"(
    comment_id SERIAL PRIMARY KEY,
    comment_content TEXT NOT NULL,
    created_at TIMESTAMP NOT NULL,
    task_id INT,
    FOREIGN KEY (task_id) REFERENCES Task(task_id)
);

CREATE TABLE CommentRelation(
    child_id INT PRIMARY KEY,
    parent_id INT,
    FOREIGN KEY (child_id) REFERENCES "Comment"(comment_id),
    FOREIGN KEY (parent_id) REFERENCES "Comment"(comment_id)
);

CREATE TABLE ProjectRole(
    user_id INT,
    project_id INT,
    user_role RoleTypes DEFAULT 'ProjectMember',
    PRIMARY KEY (user_id, project_id),
    FOREIGN KEY (user_id) REFERENCES "User"(user_id),
    FOREIGN KEY (project_id) REFERENCES Project(project_id)
);

CREATE TABLE FavouriteProject(
    user_id INT,
```

```sql
    project_id INT,
    PRIMARY KEY (user_id, project_id),
    FOREIGN KEY (user_id) REFERENCES "User"(user_id),
    FOREIGN KEY (project_id) REFERENCES Project(project_id)
);

CREATE TABLE RoleChangeNotification(
    role_change_notification_id SERIAL PRIMARY KEY,
    created_at TIMESTAMP NOT NULL,
    updated_user INT NOT NULL,
    updated_project INT NOT NULL,
    updated_role RoleTypes,
    FOREIGN KEY (updated_user, updated_project) REFERENCES ProjectRole(user_id, project_id)
);

CREATE TABLE InviteNotification(
    invite_notification_id SERIAL PRIMARY KEY,
    created_at TIMESTAMP NOT NULL,
    user_id INT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE TaskNotification(
    task_notification_id SERIAL PRIMARY KEY,
    created_at TIMESTAMP NOT NULL,
    task_id INT NOT NULL,
    notify_type TaskNotificationType NOT NULL,
    FOREIGN KEY (task_id) REFERENCES Task(task_id)
);

CREATE TABLE Image(
    image_id SERIAL PRIMARY KEY,
    image_path VARCHAR(255) NOT NULL,
    user_id INT UNIQUE NOT NULL,
    FOREIGN KEY (user_id) REFERENCES "User"(user_id)
);

CREATE TABLE ProjectTaskStatus(
    task_status_id INT NOT NULL,
    project_id INT NOT NULL,
    PRIMARY KEY (task_status_id, project_id),
    FOREIGN KEY (task_status_id) REFERENCES TaskStatus(task_status_id),
    FOREIGN KEY (project_id) REFERENCES Project(project_id)
);

-- Indexes

CREATE INDEX idx_task_deadline ON Task USING btree (deadline);
CLUSTER Task USING idx_task_deadline;

CREATE INDEX idx_task_status ON Task USING btree (task_status_id);

CREATE INDEX idx_project_status ON Project USING btree (project_status);
CLUSTER Project USING idx_project_status;

ALTER TABLE Task
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION task_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.task_title), 'A') ||
         setweight(to_tsvector('english', NEW.task_description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
```

```
         IF (NEW.task_title <> OLD.task_title OR NEW.task_description <> OLD.task_description) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.task_title), 'A') ||
             setweight(to_tsvector('english', NEW.task_description), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER task_search_update
 BEFORE INSERT OR UPDATE ON Task
 FOR EACH ROW
 EXECUTE PROCEDURE task_search_update();

CREATE INDEX task_search_idx ON Task USING GIN (tsvectors);

ALTER TABLE Project
ADD COLUMN tsvectors TSVECTOR;

CREATE FUNCTION project_search_update() RETURNS TRIGGER AS $$
BEGIN
 IF TG_OP = 'INSERT' THEN
        NEW.tsvectors = (
         setweight(to_tsvector('english', NEW.project_name), 'A') ||
         setweight(to_tsvector('english', NEW.project_description), 'B')
        );
 END IF;
 IF TG_OP = 'UPDATE' THEN
         IF (NEW.project_name <> OLD.project_name OR NEW.project_description <> OLD.project_description) THEN
           NEW.tsvectors = (
             setweight(to_tsvector('english', NEW.project_name), 'A') ||
             setweight(to_tsvector('english', NEW.project_description), 'B')
           );
         END IF;
 END IF;
 RETURN NEW;
END $$
LANGUAGE plpgsql;

CREATE TRIGGER project_search_update
 BEFORE INSERT OR UPDATE ON Project
 FOR EACH ROW
 EXECUTE PROCEDURE project_search_update();

CREATE INDEX project_search_idx ON Project USING GIN (tsvectors);

-- Triggers

CREATE OR REPLACE FUNCTION notify_task_completion() RETURNS TRIGGER AS $$
DECLARE
    completed_status_id INT;
BEGIN
    SELECT task_status_id INTO completed_status_id
    FROM TaskStatus
    WHERE task_status_name = 'Completed';

    IF NEW.task_status_id = completed_status_id AND OLD.task_status_id IS DISTINCT FROM NEW.task_status_id THEN
        INSERT INTO TaskNotification (created_at, task_id, notify_type)
        VALUES (NOW(), NEW.task_id, 'TaskCompleted');
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER task_completion_trigger
AFTER UPDATE OF task_status_id ON Task
FOR EACH ROW
EXECUTE FUNCTION notify_task_completion();

CREATE OR REPLACE FUNCTION notify_coordinator_change() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO RoleChangeNotification (created_at, updated_user, updated_project, updated_role)
    VALUES (NOW(), NEW.user_id, NEW.project_id, NEW.user_role);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER coordinator_change_trigger
AFTER UPDATE OF user_role ON ProjectRole
FOR EACH ROW
WHEN (NEW.user_role = 'ProjectCoordinator')
EXECUTE FUNCTION notify_coordinator_change();

CREATE OR REPLACE FUNCTION notify_task_assignment() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'NewAssignment');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_assignment_trigger
AFTER INSERT ON AssignedTask
FOR EACH ROW
EXECUTE FUNCTION notify_task_assignment();

CREATE OR REPLACE FUNCTION notify_invitation_creation() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO InviteNotification (created_at, user_id)
    VALUES (NOW(), NEW.receiver_id);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER invitation_creation_trigger
AFTER INSERT ON Invitation
FOR EACH ROW
EXECUTE FUNCTION notify_invitation_creation();

CREATE OR REPLACE FUNCTION notify_task_comment() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'TaskComment');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER task_comment_trigger
AFTER INSERT ON "Comment"
FOR EACH ROW
EXECUTE FUNCTION notify_task_comment();

CREATE OR REPLACE FUNCTION notify_deadline_change() RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO TaskNotification (created_at, task_id, notify_type)
    VALUES (NOW(), NEW.task_id, 'DeadlineChanged');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER deadline_change_trigger
```

```
AFTER UPDATE OF deadline ON Task
FOR EACH ROW
WHEN (OLD.deadline IS DISTINCT FROM NEW.deadline)
EXECUTE FUNCTION notify_deadline_change();
```

A.2. Database population

In this section, we present a sample of the Planora database population script, demonstrating how data is inserted into the database tables. For illustration purposes, only the first 10 lines of the script are included here.

```
INSERT INTO "User" (username, user_password, email, biography, user_status, last_login, created_at, is_admin)
VALUES
    ('user1', md5('password1'), 'user1@example.com', 'Bio for user1', 'Active', NOW(), NOW(), TRUE),
    ('user2', md5('password2'), 'user2@example.com', 'Bio for user2', 'Active', NOW(), NOW(), FALSE),
    ('user3', md5('password3'), 'user3@example.com', 'Bio for user3', 'Inactive', NOW() - INTERVAL '5 days', NOW(), FAL
    ('user4', md5('password4'), 'user4@example.com', 'Bio for user4', 'Active', NOW(), NOW(), FALSE),
    ('user5', md5('password5'), 'user5@example.com', 'Bio for user5', 'Blocked', NOW() - INTERVAL '2 days', NOW(), FALS
    ('user6', md5('password6'), 'user6@example.com', 'Bio for user6', 'Banned', NOW() - INTERVAL '10 days', NOW(), FALS
    ('user7', md5('password7'), 'user7@example.com', 'Bio for user7', 'Active', NOW(), NOW(), FALSE),
    ('user8', md5('password8'), 'user8@example.com', 'Bio for user8', 'Active', NOW(), NOW(), FALSE),
    ('user9', md5('password9'), 'user9@example.com', 'Bio for user9', 'Inactive', NOW() - INTERVAL '3 days', NOW(), FAL
    ('user10', md5('password10'), 'user10@example.com', 'Bio for user10', 'Active', NOW(), NOW(), FALSE);

INSERT INTO TaskStatus (task_status_name)
VALUES
    ('Not Started'),
    ('In Progress'),
    ('Completed'),
    ('On Hold'),
    ('Cancelled'),
    ('Reviewing'),
    ('Deferred'),
    ('Blocked'),
    ('Awaiting Approval'),
    ('Ready for Deployment');

INSERT INTO Project (project_name, project_description, created_at, project_status, creator_id)
VALUES
    ('Project A', 'Description A', NOW(), 'Active', 1),
    ('Project B', 'Description B', NOW(), 'Completed', 2),
    ('Project C', 'Description C', NOW(), 'Archived', 3),
    ('Project D', 'Description D', NOW(), 'OnHold', 4),
    ('Project E', 'Description E', NOW(), 'Active', 5),
    ('Project F', 'Description F', NOW(), 'Completed', 6),
    ('Project G', 'Description G', NOW(), 'Archived', 7),
    ('Project H', 'Description H', NOW(), 'OnHold', 8),
    ('Project I', 'Description I', NOW(), 'Active', 9),
    ('Project J', 'Description J', NOW(), 'Active', 10);

INSERT INTO Invitation (invitation_message, sender_id, receiver_id, project_id)
VALUES
    ('Join Project A', 1, 2, 1),
    ('Join Project B', 2, 3, 2),
    ('Join Project C', 3, 4, 3),
    ('Join Project D', 4, 5, 4),
    ('Join Project E', 5, 6, 5),
    ('Join Project F', 6, 7, 6),
    ('Join Project G', 7, 8, 7),
    ('Join Project H', 8, 9, 8),
    ('Join Project I', 9, 10, 9),
    ('Join Project J', 10, 1, 10);

INSERT INTO Task (task_title, task_description, created_at, deadline, task_status_id, task_priority, project_id, user_i
VALUES
    ('Task 1', 'Description 1', NOW(), NOW() + INTERVAL '7 days', 1, 'High', 1, 1),
    ('Task 2', 'Description 2', NOW(), NOW() + INTERVAL '8 days', 2, 'Medium', 2, 2),
```

```sql
    ('Task 3', 'Description 3', NOW(), NOW() + INTERVAL '9 days', 3, 'Low', 3, 3),
    ('Task 4', 'Description 4', NOW(), NOW() + INTERVAL '10 days', 4, 'High', 4, 4),
    ('Task 5', 'Description 5', NOW(), NOW() + INTERVAL '11 days', 5, 'Medium', 5, 5),
    ('Task 6', 'Description 6', NOW(), NOW() + INTERVAL '12 days', 6, 'Low', 6, 6),
    ('Task 7', 'Description 7', NOW(), NOW() + INTERVAL '13 days', 7, 'High', 7, 7),
    ('Task 8', 'Description 8', NOW(), NOW() + INTERVAL '14 days', 8, 'Medium', 8, 8),
    ('Task 9', 'Description 9', NOW(), NOW() + INTERVAL '15 days', 9, 'Low', 9, 9),
    ('Task 10', 'Description 10', NOW(), NOW() + INTERVAL '16 days', 10, 'High', 10, 10);

INSERT INTO AssignedTask (user_id, task_id)
VALUES
    (1, 1),
    (2, 2),
    (3, 3),
    (4, 4),
    (5, 5),
    (6, 6),
    (7, 7),
    (8, 8),
    (9, 9),
    (10, 10);

INSERT INTO "Comment" (comment_content, created_at, task_id)
VALUES
    ('Comment 1', NOW(), 1),
    ('Comment 2', NOW(), 2),
    ('Comment 3', NOW(), 3),
    ('Comment 4', NOW(), 4),
    ('Comment 5', NOW(), 5),
    ('Comment 6', NOW(), 6),
    ('Comment 7', NOW(), 7),
    ('Comment 8', NOW(), 8),
    ('Comment 9', NOW(), 9),
    ('Comment 10', NOW(), 10);

INSERT INTO CommentRelation (child_id, parent_id)
VALUES
    (2, 1),
    (4, 3),
    (6, 5),
    (8, 7),
    (10, 9);

INSERT INTO ProjectRole (user_id, project_id, user_role)
VALUES
    (1, 1, 'ProjectCoordinator'),
    (2, 2, 'ProjectCoordinator'),
    (3, 3, 'ProjectMember'),
    (4, 4, 'ProjectMember'),
    (5, 5, 'ProjectCoordinator'),
    (6, 6, 'ProjectCoordinator'),
    (7, 7, 'ProjectMember'),
    (8, 8, 'ProjectMember'),
    (9, 9, 'ProjectCoordinator'),
    (10, 10, 'ProjectMember');

INSERT INTO FavouriteProject (user_id, project_id)
VALUES
    (1, 1),
    (2, 2),
    (3, 3),
    (4, 4),
    (5, 5),
    (6, 6),
    (7, 7),
    (8, 8),
    (9, 9),
    (10, 10);

INSERT INTO RoleChangeNotification (created_at, updated_user, updated_project, updated_role)
VALUES
    (NOW() - INTERVAL '1 day', 1, 1, 'ProjectCoordinator'),
    (NOW() - INTERVAL '2 days', 2, 2, 'ProjectCoordinator'),
    (NOW() - INTERVAL '3 days', 3, 3, 'ProjectMember'),
    (NOW() - INTERVAL '4 days', 4, 4, 'ProjectMember'),
    (NOW() - INTERVAL '5 days', 5, 5, 'ProjectCoordinator'),
    (NOW() - INTERVAL '6 days', 6, 6, 'ProjectCoordinator'),
    (NOW() - INTERVAL '7 days', 7, 7, 'ProjectMember'),
    (NOW() - INTERVAL '8 days', 8, 8, 'ProjectMember'),
    (NOW() - INTERVAL '9 days', 9, 9, 'ProjectCoordinator'),
    (NOW() - INTERVAL '10 days', 10, 10, 'ProjectMember');

INSERT INTO InviteNotification (created_at, user_id)
VALUES
    (NOW() - INTERVAL '1 day', 1),
    (NOW() - INTERVAL '2 days', 2),
    (NOW() - INTERVAL '3 days', 3),
    (NOW() - INTERVAL '4 days', 4),
    (NOW() - INTERVAL '5 days', 5),
    (NOW() - INTERVAL '6 days', 6),
    (NOW() - INTERVAL '7 days', 7),
    (NOW() - INTERVAL '8 days', 8),
    (NOW() - INTERVAL '9 days', 9),
    (NOW() - INTERVAL '10 days', 10);

INSERT INTO TaskNotification (created_at, task_id, notify_type)
VALUES
    (NOW() - INTERVAL '1 day', 1, 'DeadlineChanged'),
    (NOW() - INTERVAL '2 days', 2, 'TaskPriorityChanged'),
    (NOW() - INTERVAL '3 days', 3, 'NewAssignment'),
    (NOW() - INTERVAL '4 days', 4, 'TaskCompleted'),
    (NOW() - INTERVAL '5 days', 5, 'TaskComment'),
    (NOW() - INTERVAL '6 days', 6, 'DeadlineChanged'),
    (NOW() - INTERVAL '7 days', 7, 'TaskPriorityChanged'),
    (NOW() - INTERVAL '8 days', 8, 'NewAssignment'),
    (NOW() - INTERVAL '9 days', 9, 'TaskCompleted'),
    (NOW() - INTERVAL '10 days', 10, 'TaskComment');

INSERT INTO Image (image_path, user_id)
VALUES
    ('image1.jpg', 1),
    ('image2.jpg', 2),
    ('image3.jpg', 3),
    ('image4.jpg', 4),
    ('image5.jpg', 5),
    ('image6.jpg', 6),
    ('image7.jpg', 7),
    ('image8.jpg', 8),
    ('image9.jpg', 9),
    ('image10.jpg', 10);

INSERT INTO ProjectTaskStatus (task_status_id, project_id)
VALUES
    (1, 1),
    (2, 2),
    (3, 3),
    (4, 4),
    (5, 5),
    (6, 6),
    (7, 7),
    (8, 8),
    (9, 9),
    (10, 10);
```

Revision history

Changes made to the first submission:

1. Item 1
2. ..

GROUP24135, 07/11/2024

- Diogo Ferreira, up202205295@up.pt
- Gonçalo Marques, up202206205@up.pt
- Lucas Greco, up202208296@up.pt
- Rui Cruz, up202208011@up.pt **(editor)**