

Procedures, Functions and Triggers

Кристијан Тримчески 231132

Елена Спасовска 231141

Марија Сергиевска 231048

Procedures:

1. submit_offer - Allows a worker to submit a price offer on an open task request, or enables a client to initiate an offer on behalf of a worker. Performs full validation before inserting the offer record and dispatches a notification to the client. Used when a worker or a client submits an offer.

```
CREATE OR REPLACE PROCEDURE submit_offer(
    p_worker_id      INT,
    p_task_request_id INT,
    p_price          INT,
    p_initiated_by   VARCHAR(10)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status      VARCHAR(20);
    v_category_id      INT;
    v_client_user_id   INT;
    v_offer_id         INT;
    v_notif_type_id    INT;
    v_duplicate        INT;
BEGIN

    IF p_initiated_by NOT IN ('CLIENT', 'WORKER') THEN
        RAISE EXCEPTION 'initiated_by must be CLIENT or WORKER, got: %',
p_initiated_by;
    END IF;

    IF p_price <= 0 THEN
        RAISE EXCEPTION 'Price must be greater than 0, got: %', p_price;
    END IF;

    SELECT status, category_id
    INTO v_task_status, v_category_id
    FROM TaskRequest
    WHERE id = p_task_request_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'TaskRequest % does not exist', p_task_request_id;
    END IF;

    IF v_task_status <> 'OPEN' THEN
        RAISE EXCEPTION 'TaskRequest % is not open (status: %)',
p_task_request_id, v_task_status;
    END IF;

    IF NOT EXISTS (
        SELECT 1 FROM WorkerCategory
        WHERE worker_id = p_worker_id
        AND category_id = v_category_id
```

```

    ) THEN
        RAISE EXCEPTION 'Worker % is not registered for the required category',
p_worker_id;
    END IF;

    SELECT COUNT(*) INTO v_duplicate
    FROM Offer
    WHERE worker_id = p_worker_id
        AND task_request_id = p_task_request_id
        AND offer_status IN ('PENDING', 'ACCEPTED');

    IF v_duplicate > 0 THEN
        RAISE EXCEPTION 'Worker % already has an active offer for TaskRequest
%', p_worker_id, p_task_request_id;
    END IF;

    INSERT INTO Offer (worker_id, task_request_id, price, offer_status,
initiated by)
    VALUES (p_worker_id, p_task_request_id, p_price, 'PENDING', p_initiated_by)
    RETURNING id INTO v_offer_id;

    SELECT u.id
    INTO v_client_user_id
    FROM TaskRequest tr
    JOIN Client c ON tr.client_id = c.id
    JOIN UserAccount u ON c.user_id = u.id
    WHERE tr.id = p_task_request_id;

    SELECT id INTO v_notif_type_id
    FROM NotificationType
    WHERE name = 'NEW_OFFER'
    LIMIT 1;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id)
    VALUES (
        'New Offer Received',
        'A worker has submitted an offer for your task request #' ||
p_task_request_id,
        v_client_user_id,
        v_notif_type_id,
        v_offer_id
    );

    RAISE NOTICE 'Offer % submitted successfully for TaskRequest %', v_offer_id,
p_task_request_id;
END;
$$;

```

2. `accept_offer` - Accepts a specific pending offer, rejects all competing offers on the same task request, closes the task request, creates an active Task, and notifies both the accepted worker and all rejected workers. Used when a user accepts an offer.

```

CREATE OR REPLACE PROCEDURE accept_offer(
    p_offer_id INT
)
LANGUAGE plpgsql

```

```

AS $$
DECLARE
    v_task_request_id INT;
    v_worker_id       INT;
    v_offer_status    VARCHAR(20);
    v_task_id         INT;
    v_notif_type_acc  INT;
    v_notif_type_rej  INT;
    v_worker_user_id  INT;
    rec               RECORD;
BEGIN

    SELECT task_request_id, worker_id, offer_status
    INTO v_task_request_id, v_worker_id, v_offer_status
    FROM Offer
    WHERE id = p_offer_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Offer % does not exist', p_offer_id;
    END IF;

    IF v_offer_status <> 'PENDING' THEN
        RAISE EXCEPTION 'Offer % is not pending (status: %)', p_offer_id,
v_offer_status;
    END IF;

    UPDATE Offer
    SET offer_status = 'ACCEPTED',
        updated_at   = CURRENT_TIMESTAMP
    WHERE id = p_offer_id;

    UPDATE Offer
    SET offer_status = 'REJECTED',
        updated_at   = CURRENT_TIMESTAMP
    WHERE task_request_id = v_task_request_id
        AND id <> p_offer_id
        AND offer_status = 'PENDING';

    UPDATE TaskRequest
    SET status      = 'CLOSED',
        updated_at = CURRENT_TIMESTAMP
    WHERE id = v_task_request_id;

    INSERT INTO Task (offer_id, status)
    VALUES (p_offer_id, 'ACTIVE')
    RETURNING id INTO v_task_id;

    SELECT id INTO v_notif_type_acc FROM NotificationType WHERE name =
'OFFER_ACCEPTED' LIMIT 1;
    SELECT id INTO v_notif_type_rej FROM NotificationType WHERE name =
'OFFER_REJECTED' LIMIT 1;

    SELECT u.id INTO v_worker_user_id
    FROM Worker w
    JOIN UserAccount u ON w.user_id = u.id
    WHERE w.id = v_worker_id;

```

```

INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id, task_id)
VALUES (
    'Your Offer Was Accepted!',
    'Congratulations! Your offer #' || p_offer_id || ' has been accepted.
Task #' || v_task_id || ' is now active.',
    v_worker_user_id,
    v_notif_type_acc,
    p_offer_id,
    v_task_id
);

FOR rec IN
SELECT w.user_id AS uid, o.id AS oid
FROM Offer o
JOIN Worker w ON o.worker_id = w.id
WHERE o.task_request_id = v_task_request_id
      AND o.id <> p_offer_id
      AND o.offer_status = 'REJECTED'
LOOP
    INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id)
    VALUES (
        'Offer Not Selected',
        'Unfortunately your offer #' || rec.oid || ' was not selected for
this task.',
        rec.uid,
        v_notif_type_rej,
        rec.oid
    );
END LOOP;

RAISE NOTICE 'Offer % accepted. Task % created.', p_offer_id, v_task_id;
END;
$$;

```

3. complete_task - Marks an active task as completed, creates a pending payment record for the agreed offer price, links it via TaskPayment, and notifies both the client and the worker. Used when a task is finished.

```

CREATE OR REPLACE PROCEDURE complete_task(
    p_task_id          INT,
    p_payment_method   VARCHAR(50)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status      VARCHAR(20);
    v_offer_id         INT;
    v_offer_price      INT;
    v_worker_id        INT;
    v_client_id        INT;
    v_client_user_id   INT;
    v_worker_user_id   INT;
    v_payment_id       INT;
    v_notif_type_id    INT;

```

BEGIN

```
    IF p_payment_method NOT IN ('CARD', 'CASH', 'PAYPAL') THEN
        RAISE EXCEPTION 'Invalid payment method: %. Must be CARD, CASH, or
PAYPAL', p_payment_method;
    END IF;
```

```
    SELECT status, offer_id
    INTO v_task_status, v_offer_id
    FROM Task
    WHERE id = p_task_id;
```

```
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;
```

```
    IF v_task_status <> 'ACTIVE' THEN
        RAISE EXCEPTION 'Task % is not active (status: %)', p_task_id,
v_task_status;
    END IF;
```

```
    SELECT o.price, o.worker_id, tr.client_id
    INTO v_offer_price, v_worker_id, v_client_id
    FROM Offer o
    JOIN TaskRequest tr ON o.task_request_id = tr.id
    WHERE o.id = v_offer_id;
```

```
    UPDATE Task
    SET status          = 'COMPLETED',
        completed_at = CURRENT_TIMESTAMP,
        updated_at    = CURRENT_TIMESTAMP
    WHERE id = p_task_id;
```

```
    INSERT INTO Payment (amount, payment_method, status, task_id, client_id,
worker_id)
    VALUES (v_offer_price, p_payment_method, 'PENDING', p_task_id, v_client_id,
v_worker_id)
    RETURNING id INTO v_payment_id;
```

```
    INSERT INTO TaskPayment (payment_id, task_id)
    VALUES (v_payment_id, p_task_id);
```

```
    SELECT id INTO v_notif_type_id FROM NotificationType WHERE name =
'TASK_COMPLETED' LIMIT 1;
```

```
    SELECT u.id INTO v_client_user_id
    FROM Client c JOIN UserAccount u ON c.user_id = u.id
    WHERE c.id = v_client_id;
```

```
    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id, payment_id)
    VALUES (
        'Task Completed',
        'Task #' || p_task_id || ' has been marked as completed. Payment of ' ||
v_offer_price || ' is pending.',
```

```

        v_client_user_id,
        v_notif_type_id,
        p_task_id,
        v_payment_id
    );

    SELECT u.id INTO v_worker_user_id
    FROM Worker w JOIN UserAccount u ON w.user_id = u.id
    WHERE w.id = v_worker_id;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id, payment_id)
    VALUES (
        'Task Completed - Payment Incoming',
        'Task #' || p_task_id || ' is complete. You will receive payment of ' ||
v_offer_price || ' shortly.',
        v_worker_user_id,
        v_notif_type_id,
        p_task_id,
        v_payment_id
    );

    RAISE NOTICE 'Task % completed. Payment % created.', p_task_id,
v_payment_id;
END;
$$;

```

4. cancel_task - Cancels an active task, marks any pending payment as FAILED, and notifies both the client and the worker with an optional reason string. Used when a user cancels the active task.

```

CREATE OR REPLACE PROCEDURE cancel_task(
    p_task_id INT,
    p_reason VARCHAR(255)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status VARCHAR(20);
    v_offer_id INT;
    v_worker_id INT;
    v_client_id INT;
    v_client_user_id INT;
    v_worker_user_id INT;
    v_notif_type_id INT;
    v_payment_id INT;
BEGIN

    SELECT status, offer_id
    INTO v_task_status, v_offer_id
    FROM Task
    WHERE id = p_task_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;

    IF v_task_status <> 'ACTIVE' THEN
        RAISE EXCEPTION 'Task % cannot be cancelled (status: %)', p_task_id,
v_task_status;
    END IF;

```

```
END IF;
```

```
SELECT o.worker_id, tr.client_id
INTO v_worker_id, v_client_id
FROM Offer o
JOIN TaskRequest tr ON o.task_request_id = tr.id
WHERE o.id = v_offer_id;
```

```
UPDATE Task
SET status = 'CANCELLED',
    updated_at = CURRENT_TIMESTAMP
WHERE id = p_task_id;
```

```
UPDATE Payment
SET status = 'FAILED',
    updated_at = CURRENT_TIMESTAMP
WHERE task_id = p_task_id
    AND status = 'PENDING'
RETURNING id INTO v_payment_id;
```

```
SELECT id INTO v_notif_type_id
FROM NotificationType
WHERE name = 'TASK_CANCELLED'
LIMIT 1;
```

```
SELECT u.id INTO v_client_user_id
FROM Client c
JOIN UserAccount u ON c.user_id = u.id
WHERE c.id = v_client_id;
```

```
INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
VALUES (
    'Task Cancelled',
    'Task #' || p_task_id || ' has been cancelled. Reason: ' ||
COALESCE(p_reason, 'No reason provided'),
    v_client_user_id,
    v_notif_type_id,
    p_task_id
);
```

```
SELECT u.id INTO v_worker_user_id
FROM Worker w
JOIN UserAccount u ON w.user_id = u.id
WHERE w.id = v_worker_id;
```

```
INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
VALUES (
    'Task Cancelled',
    'Task #' || p_task_id || ' has been cancelled. Reason: ' ||
COALESCE(p_reason, 'No reason provided'),
    v_worker_user_id,
    v_notif_type_id,
    p_task_id
);
```

```

        RAISE NOTICE 'Task % cancelled successfully.', p_task_id;
END;
$$;

```

5. submit_complaint - Allows a client to file a complaint against a worker on a completed task or a worker to file a complaint against a client on a completed task. Validates ownership, prevents duplicate open complaints, inserts the complaint record, and notifies the one that filed the complaint. Used when a user issues a complaint.

```

CREATE OR REPLACE PROCEDURE submit_complaint(
    p_task_id      INT,
    p_client_id    INT,
    p_worker_id    INT,
    p_reason       TEXT,
    p_description  TEXT,
    p_filed_by     VARCHAR(10)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status    VARCHAR(20);
    v_offer_id       INT;
    v_real_client_id INT;
    v_real_worker_id INT;
    v_complaint_id   INT;
    v_filer_user_id  INT;
    v_notif_type_id  INT;
BEGIN
    IF p_filed_by NOT IN ('CLIENT', 'WORKER') THEN
        RAISE EXCEPTION 'p_filed_by must be CLIENT or WORKER, got: %',
p_filed_by;
    END IF;

    SELECT status, offer_id
    INTO   v_task_status, v_offer_id
    FROM   Task
    WHERE  id = p_task_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;

    IF v_task_status <> 'COMPLETED' THEN
        RAISE EXCEPTION 'Complaints can only be filed on COMPLETED tasks
(status: %)', v_task_status;
    END IF;

    SELECT tr.client_id, o.worker_id
    INTO   v_real_client_id, v_real_worker_id
    FROM   Offer o
    JOIN   TaskRequest tr ON o.task_request_id = tr.id
    WHERE  o.id = v_offer_id;

    IF v_real_client_id <> p_client_id THEN
        RAISE EXCEPTION 'Client % is not the client for Task %', p_client_id,
p_task_id;.
    END IF;

```



```

    IF v_real_worker_id <> p_worker_id THEN
        RAISE EXCEPTION 'Worker % is not the worker for Task %', p_worker_id,
p_task_id;
    END IF;

    IF p_filed_by = 'CLIENT' AND EXISTS (
        SELECT 1 FROM Complaint
        WHERE task_id = p_task_id
        AND client_id = p_client_id
        AND status = 'OPEN'
    ) THEN
        RAISE EXCEPTION 'Client % already has an open complaint for Task %',
p_client_id, p_task_id;
    END IF;

    IF p_filed_by = 'WORKER' AND EXISTS (
        SELECT 1 FROM Complaint
        WHERE task_id = p_task_id
        AND worker_id = p_worker_id
        AND status = 'OPEN'
    ) THEN
        RAISE EXCEPTION 'Worker % already has an open complaint for Task %',
p_worker_id, p_task_id;
    END IF;

    INSERT INTO Complaint (reason, description, status, task_id, client_id,
worker_id)
    VALUES (p_reason, p_description, 'OPEN', p_task_id, p_client_id,
p_worker_id)
    RETURNING id INTO v_complaint_id;

    IF p_filed_by = 'CLIENT' THEN
        SELECT u.id INTO v_filer_user_id
        FROM Client c
        JOIN UserAccount u ON c.user_id = u.id
        WHERE c.id = p_client_id;
    ELSE
        SELECT u.id INTO v_filer_user_id
        FROM Worker w
        JOIN UserAccount u ON w.user_id = u.id
        WHERE w.id = p_worker_id;
    END IF;

    SELECT id INTO v_notif_type_id
    FROM NotificationType
    WHERE name = 'COMPLAINT_FILED'
    LIMIT 1;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
    VALUES (
        'Complaint Submitted',
        'Your complaint for Task #' || p_task_id || ' has been received and is
under review. Reason: ' || p_reason,
        v_filer_user_id,
        v_notif_type_id,
        p_task_id
    );

    RAISE NOTICE 'Complaint % filed for Task % by %.', v_complaint_id,
p_task_id, p_filed_by;

```

```
END;  
$$;
```

6. register_client - Registers a new client in the marketplace. Validates name, email format, phone format, uniqueness of email/phone, and coordinate ranges. Upserts the location and creates the user account, client record, and default push notification preferences. Used for new client sign-up.

```
CREATE OR REPLACE PROCEDURE register_client(  
    p_name      VARCHAR(100),  
    p_surname   VARCHAR(100),  
    p_email     VARCHAR(255),  
    p_phone     VARCHAR(20),  
    p_city      VARCHAR(100),  
    p_latitude   DECIMAL(9,6),  
    p_longitude  DECIMAL(9,6)  
)  
LANGUAGE plpgsql  
AS $$  
DECLARE  
    v_user_id    INT;  
    v_location_id INT;  
BEGIN  
  
    IF p_name IS NULL OR TRIM(p_name) = '' THEN  
        RAISE EXCEPTION 'Name cannot be empty';  
    END IF;  
    IF p_surname IS NULL OR TRIM(p_surname) = '' THEN  
        RAISE EXCEPTION 'Surname cannot be empty';  
    END IF;  
  
    IF p_email !~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' THEN  
        RAISE EXCEPTION 'Invalid email format: %', p_email;  
    END IF;  
  
    IF p_phone !~ '^\+?[0-9]{7,15}$' THEN  
        RAISE EXCEPTION 'Invalid phone format: %', p_phone;  
    END IF;  
  
    IF p_latitude NOT BETWEEN -90 AND 90 THEN  
        RAISE EXCEPTION 'Invalid latitude: %', p_latitude;  
    END IF;  
    IF p_longitude NOT BETWEEN -180 AND 180 THEN  
        RAISE EXCEPTION 'Invalid longitude: %', p_longitude;  
    END IF;  
  
    IF EXISTS (SELECT 1 FROM UserAccount WHERE email = p_email) THEN  
        RAISE EXCEPTION 'Email already registered: %', p_email;  
    END IF;  
    IF EXISTS (SELECT 1 FROM UserAccount WHERE phone_number = p_phone) THEN  
        RAISE EXCEPTION 'Phone number already registered: %', p_phone;  
    END IF;
```

```

INSERT INTO Location (city, latitude, longitude)
VALUES (p_city, p_latitude, p_longitude)
ON CONFLICT ON CONSTRAINT uq_location_cords DO NOTHING;

SELECT id INTO v_location_id
FROM Location
WHERE latitude = p_latitude AND longitude = p_longitude;

INSERT INTO UserAccount (name, surname, email, phone_number)
VALUES (p_name, p_surname, p_email, p_phone)
RETURNING id INTO v_user_id;

INSERT INTO Client (user_id)
VALUES (v_user_id);

INSERT INTO NotificationPreference (is_enabled, channel, user_id,
notification_type_id)
SELECT TRUE, 'PUSH', v_user_id, nt.id
FROM NotificationType nt;

RAISE NOTICE 'Client registered successfully. user_id = %', v_user_id;
END;
$$;

```

7. register_worker - Registers a new worker with category specializations. Validates all inputs including work mode, service radius, coordinate ranges, and verifies each category exists before linking them. Upserts the location, creates the user account, worker record, and default push notification preferences. Used for new worker sign-up.

```

CREATE OR REPLACE PROCEDURE register_worker(
    p_name          VARCHAR(100),
    p_surname       VARCHAR(100),
    p_email         VARCHAR(255),
    p_phone         VARCHAR(20),
    p_work_mode     VARCHAR(20),
    p_radius_km     INT,
    p_city          VARCHAR(100),
    p_latitude      DECIMAL(9,6),
    p_longitude     DECIMAL(9,6),
    p_category_ids  INT[]
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_user_id      INT;
    v_worker_id    INT;
    v_location_id  INT;
    v_cat_id       INT;

```

BEGIN

IF p_name **IS NULL OR TRIM**(p_name) = '' **THEN**
 RAISE EXCEPTION 'Name cannot be empty';

END IF;

IF p_surname **IS NULL OR TRIM**(p_surname) = '' **THEN**
 RAISE EXCEPTION 'Surname cannot be empty';

END IF;

IF p_email !~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\$' **THEN**
 RAISE EXCEPTION 'Invalid email format: %', p_email;

END IF;

IF p_phone !~ '^\\+?[0-9]{7,15}\$' **THEN**
 RAISE EXCEPTION 'Invalid phone format: %', p_phone;

END IF;

IF p_work_mode **NOT IN** ('HYBRID', 'ONSITE', 'REMOTE') **THEN**
 RAISE EXCEPTION 'Invalid work_mode: %. Must be HYBRID, ONSITE, or
REMOTE', p_work_mode;

END IF;

IF p_radius_km <= 0 **THEN**
 RAISE EXCEPTION 'service_radius_km must be greater than 0, got: %',
p_radius_km;

END IF;

IF p_latitude **NOT BETWEEN** -90 **AND** 90 **THEN**
 RAISE EXCEPTION 'Invalid latitude: %', p_latitude;

END IF;

IF p_longitude **NOT BETWEEN** -180 **AND** 180 **THEN**
 RAISE EXCEPTION 'Invalid longitude: %', p_longitude;

END IF;

IF p_category_ids **IS NULL OR array_length**(p_category_ids, 1) = 0 **THEN**
 RAISE EXCEPTION 'Worker must have at least one category';

END IF;

IF EXISTS (SELECT 1 **FROM** UserAccount **WHERE** email = p_email) **THEN**
 RAISE EXCEPTION 'Email already registered: %', p_email;

END IF;

IF EXISTS (SELECT 1 **FROM** UserAccount **WHERE** phone_number = p_phone) **THEN**
 RAISE EXCEPTION 'Phone number already registered: %', p_phone;

END IF;

FOREACH v_cat_id **IN ARRAY** p_category_ids **LOOP**
 IF NOT EXISTS (SELECT 1 **FROM** Category **WHERE** id = v_cat_id) **THEN**
 RAISE EXCEPTION 'Category % does not exist', v_cat_id;
 END IF;
END LOOP;

INSERT INTO Location (city, latitude, longitude)
VALUES (p_city, p_latitude, p_longitude)
ON CONFLICT ON CONSTRAINT uq_location_cords **DO NOTHING;**

```

SELECT id INTO v_location_id
FROM Location
WHERE latitude = p_latitude AND longitude = p_longitude;

INSERT INTO UserAccount (name, surname, email, phone_number)
VALUES (p_name, p_surname, p_email, p_phone)
RETURNING id INTO v_user_id;

INSERT INTO Worker (user_id, work_mode, service_radius_km, location_id)
VALUES (v_user_id, p_work_mode, p_radius_km, v_location_id)
RETURNING id INTO v_worker_id;

FOREACH v_cat_id IN ARRAY p_category_ids LOOP
    INSERT INTO WorkerCategory (worker_id, category_id)
    VALUES (v_worker_id, v_cat_id);
END LOOP;

INSERT INTO NotificationPreference (is_enabled, channel, user_id,
notification_type_id)
SELECT TRUE, 'PUSH', v_user_id, nt.id
FROM NotificationType nt;

RAISE NOTICE 'Worker registered successfully. user_id = %, worker_id = %',
v_user_id, v_worker_id;
END;
$$;

```

8. create_task_request - Creates a new task request from a client seeking services. Validates the client exists, description is non-empty, work mode is valid, category exists, and coordinates are in range. Upserts the task location and inserts the request with OPEN status. Used when a client posts a new task.

```

CREATE OR REPLACE PROCEDURE create_task_request(
    p_client_id INT,
    p_description VARCHAR(1000),
    p_work_mode VARCHAR(20),
    p_category_id INT,
    p_city VARCHAR(100),
    p_latitude DECIMAL(9,6),
    p_longitude DECIMAL(9,6)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_location_id INT;
    v_task_req_id INT;
    v_open_count INT;
BEGIN

```

```

IF NOT EXISTS (SELECT 1 FROM Client WHERE id = p_client_id) THEN
    RAISE EXCEPTION 'Client % does not exist', p_client_id;
END IF;

IF p_description IS NULL OR TRIM(p_description) = '' THEN
    RAISE EXCEPTION 'Description cannot be empty';
END IF;

IF p_work_mode NOT IN ('HYBRID', 'ONSITE', 'REMOTE') THEN
    RAISE EXCEPTION 'Invalid work_mode: %. Must be HYBRID, ONSITE, or
REMOTE', p_work_mode;
END IF;

IF NOT EXISTS (SELECT 1 FROM Category WHERE id = p_category_id) THEN
    RAISE EXCEPTION 'Category % does not exist', p_category_id;
END IF;

IF p_latitude NOT BETWEEN -90 AND 90 THEN
    RAISE EXCEPTION 'Invalid latitude: %', p_latitude;
END IF;
IF p_longitude NOT BETWEEN -180 AND 180 THEN
    RAISE EXCEPTION 'Invalid longitude: %', p_longitude;
END IF;

INSERT INTO Location (city, latitude, longitude)
VALUES (p_city, p_latitude, p_longitude)
ON CONFLICT ON CONSTRAINT uq_location_cords DO NOTHING;

SELECT id INTO v_location_id
FROM Location
WHERE latitude = p_latitude AND longitude = p_longitude;

INSERT INTO TaskRequest (client_id, description, work_mode, status,
category_id, location_id)
VALUES (p_client_id, p_description, p_work_mode, 'OPEN', p_category_id,
v_location_id)
RETURNING id INTO v_task_req_id;

RAISE NOTICE 'Task request % created successfully for client %',
v_task_req_id, p_client_id;
END;
$$;

```

9. delete_user_account - Soft-deletes a user account by setting the deleted_at timestamp. Blocks deletion if the user has active tasks, pending payments, or open complaints. Also soft-deletes any OPEN task requests and rejects any PENDING offers belonging to the user. Used for account deactivation.

```

CREATE OR REPLACE PROCEDURE delete_user_account(
    p_user_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_active_tasks_worker    INT;
    v_active_tasks_client    INT;
    v_pending_payments       INT;
    v_open_complaints        INT;
BEGIN

    IF NOT EXISTS (SELECT 1 FROM UserAccount WHERE id = p_user_id) THEN
        RAISE EXCEPTION 'UserAccount % does not exist', p_user_id;
    END IF;

    IF EXISTS (SELECT 1 FROM UserAccount WHERE id = p_user_id AND deleted_at IS
NOT NULL) THEN
        RAISE EXCEPTION 'UserAccount % is already deleted', p_user_id;
    END IF;

    SELECT COUNT(*) INTO v_active_tasks_worker
    FROM Task t
    JOIN Offer o ON o.id = t.offer_id
    JOIN Worker w ON w.id = o.worker_id
    WHERE w.user_id = p_user_id
        AND t.status = 'ACTIVE';

    IF v_active_tasks_worker > 0 THEN
        RAISE EXCEPTION 'Cannot delete account: user has % active task(s) as a
worker', v_active_tasks_worker;
    END IF;

    SELECT COUNT(*) INTO v_active_tasks_client
    FROM Task t
    JOIN Offer o ON o.id = t.offer_id
    JOIN TaskRequest tr ON tr.id = o.task_request_id
    JOIN Client c ON c.id = tr.client_id
    WHERE c.user_id = p_user_id
        AND t.status = 'ACTIVE';

    IF v_active_tasks_client > 0 THEN
        RAISE EXCEPTION 'Cannot delete account: user has % active task(s) as a
client', v_active_tasks_client;
    END IF;

    SELECT COUNT(*) INTO v_pending_payments
    FROM Payment p
    JOIN Worker w ON w.id = p.worker_id
    JOIN Client c ON c.id = p.client_id
    WHERE (w.user_id = p_user_id OR c.user_id = p_user_id)
        AND p.status = 'PENDING';

    IF v_pending_payments > 0 THEN
        RAISE EXCEPTION 'Cannot delete account: user has % pending payment(s)',
v_pending_payments;
    END IF;

```

```

SELECT COUNT(*) INTO v_open_complaints
FROM Complaint co
JOIN Client c ON c.id = co.client_id
JOIN Worker w ON w.id = co.worker_id
WHERE (c.user_id = p_user_id OR w.user_id = p_user_id)
      AND co.status = 'OPEN';

IF v_open_complaints > 0 THEN
    RAISE EXCEPTION 'Cannot delete account: user has % open complaint(s)',
v_open_complaints;
END IF;

UPDATE UserAccount
SET deleted_at = CURRENT_TIMESTAMP
WHERE id = p_user_id;

UPDATE TaskRequest tr
SET deleted_at = CURRENT_TIMESTAMP,
    updated_at = CURRENT_TIMESTAMP
FROM Client c
WHERE c.id      = tr.client_id
      AND c.user_id = p_user_id
      AND tr.status = 'OPEN'
      AND tr.deleted_at IS NULL;

UPDATE Offer o
SET offer_status = 'REJECTED',
    updated_at   = CURRENT_TIMESTAMP
FROM Worker w
WHERE w.id      = o.worker_id
      AND w.user_id = p_user_id
      AND o.offer_status = 'PENDING';

RAISE NOTICE 'UserAccount % soft deleted successfully', p_user_id;
END;
$$;

```

10. delete_worker - Permanently removes a worker record. Blocks deletion if the worker has active tasks, pending payments, or open complaints. Deactivates all worker badges and rejects all PENDING offers before performing the hard delete. Used when removing a worker from the platform.

```

CREATE OR REPLACE PROCEDURE delete_worker(
    p_worker_id INT
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_active_tasks      INT;
    v_pending_payments  INT;
    v_open_complaints   INT;
BEGIN

```



```

IF NOT EXISTS (SELECT 1 FROM Worker WHERE id = p_worker_id) THEN
    RAISE EXCEPTION 'Worker % does not exist', p_worker_id;
END IF;

SELECT COUNT(*) INTO v_active_tasks
FROM Task t
JOIN Offer o ON o.id = t.offer_id
WHERE o.worker_id = p_worker_id
    AND t.status = 'ACTIVE';

IF v_active_tasks > 0 THEN
    RAISE EXCEPTION 'Cannot delete worker %: has % active task(s)',
p_worker_id, v_active_tasks;
END IF;

SELECT COUNT(*) INTO v_pending_payments
FROM Payment
WHERE worker_id = p_worker_id
    AND status = 'PENDING';

IF v_pending_payments > 0 THEN
    RAISE EXCEPTION 'Cannot delete worker %: has % pending payment(s)',
p_worker_id, v_pending_payments;
END IF;

SELECT COUNT(*) INTO v_open_complaints
FROM Complaint
WHERE worker_id = p_worker_id
    AND status = 'OPEN';

IF v_open_complaints > 0 THEN
    RAISE EXCEPTION 'Cannot delete worker %: has % open complaint(s)',
p_worker_id, v_open_complaints;
END IF;

UPDATE WorkerBadge
SET is_active = FALSE
WHERE worker_id = p_worker_id;

UPDATE Offer
SET offer_status = 'REJECTED',
    updated_at = CURRENT_TIMESTAMP
WHERE worker_id = p_worker_id
    AND offer_status = 'PENDING';

DELETE FROM Worker WHERE id = p_worker_id;

RAISE NOTICE 'Worker % deleted successfully', p_worker_id;
END;
$$;

```

11. delete_client - Permanently removes a client record. Blocks deletion if the client has active tasks, pending payments, or open complaints. Soft-deletes all OPEN task requests before performing the hard delete. Used when removing a client from the platform.

```
REATE OR REPLACE PROCEDURE delete_client(  
    p_client_id INT  
)  
LANGUAGE plpgsql  
AS $$  
DECLARE  
    v_active_tasks      INT;  
    v_pending_payments  INT;  
    v_open_complaints   INT;  
BEGIN  
  
    IF NOT EXISTS (SELECT 1 FROM Client WHERE id = p_client_id) THEN  
        RAISE EXCEPTION 'Client % does not exist', p_client_id;  
    END IF;  
  
    SELECT COUNT(*) INTO v_active_tasks  
    FROM Task t  
    JOIN Offer o ON o.id = t.offer_id  
    JOIN TaskRequest tr ON tr.id = o.task_request_id  
    WHERE tr.client_id = p_client_id  
        AND t.status = 'ACTIVE';  
  
    IF v_active_tasks > 0 THEN  
        RAISE EXCEPTION 'Cannot delete client %: has % active task(s)',  
p_client_id, v_active_tasks;  
    END IF;  
  
    SELECT COUNT(*) INTO v_pending_payments  
    FROM Payment  
    WHERE client_id = p_client_id  
        AND status = 'PENDING';  
  
    IF v_pending_payments > 0 THEN  
        RAISE EXCEPTION 'Cannot delete client %: has % pending payment(s)',  
p_client_id, v_pending_payments;  
    END IF;  
  
    SELECT COUNT(*) INTO v_open_complaints  
    FROM Complaint  
    WHERE client_id = p_client_id  
        AND status = 'OPEN';  
  
    IF v_open_complaints > 0 THEN  
        RAISE EXCEPTION 'Cannot delete client %: has % open complaint(s)',  
p_client_id, v_open_complaints;  
    END IF;  
  
    UPDATE TaskRequest  
    SET deleted_at = CURRENT_TIMESTAMP,  
        updated_at = CURRENT_TIMESTAMP  
    WHERE client_id = p_client_id
```

```

AND status      = 'OPEN'
AND deleted_at IS NULL;

DELETE FROM Client WHERE id = p_client_id;

RAISE NOTICE 'Client % deleted successfully', p_client_id;
END;
$$;

```

12. update_user_account - lets users update their profile information partially or fully, only changing fields that are actually provided while validating any new values against the same constraints used at registration.

```

CREATE OR REPLACE PROCEDURE update_user_account(
    p_user_id      INT,
    p_name         VARCHAR(100) DEFAULT NULL,
    p_surname      VARCHAR(100) DEFAULT NULL,
    p_email        VARCHAR(255) DEFAULT NULL,
    p_phone        VARCHAR(20)  DEFAULT NULL
)
LANGUAGE plpgsql
AS $$
BEGIN

    IF NOT EXISTS (
        SELECT 1 FROM UserAccount
        WHERE id = p_user_id AND deleted_at IS NULL
    ) THEN
        RAISE EXCEPTION 'UserAccount % does not exist or is deleted', p_user_id;
    END IF;

    IF p_name IS NOT NULL AND TRIM(p_name) = '' THEN
        RAISE EXCEPTION 'Name cannot be empty';
    END IF;

    IF p_surname IS NOT NULL AND TRIM(p_surname) = '' THEN
        RAISE EXCEPTION 'Surname cannot be empty';
    END IF;

    IF p_email IS NOT NULL THEN
        IF p_email !~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$' THEN
            RAISE EXCEPTION 'Invalid email format: %', p_email;
        END IF;

        IF EXISTS (
            SELECT 1 FROM UserAccount
            WHERE email = p_email AND id != p_user_id
        ) THEN
            RAISE EXCEPTION 'Email already in use: %', p_email;
        END IF;
    END IF;
END IF;

```

```

IF p_phone IS NOT NULL THEN
    IF p_phone !~ '^\\+?[0-9]{7,15}$' THEN
        RAISE EXCEPTION 'Invalid phone format: %', p_phone;
    END IF;

    IF EXISTS (
        SELECT 1 FROM UserAccount
        WHERE phone_number = p_phone AND id != p_user_id
    ) THEN
        RAISE EXCEPTION 'Phone number already in use: %', p_phone;
    END IF;
END IF;

UPDATE UserAccount
SET
    name          = COALESCE(p_name, name),
    surname       = COALESCE(p_surname, surname),
    email         = COALESCE(p_email, email),
    phone_number  = COALESCE(p_phone, phone_number),
    updated_at    = CURRENT_TIMESTAMP
WHERE id = p_user_id;

RAISE NOTICE 'UserAccount % updated successfully', p_user_id;
END;
$$;

```

13. add_favourite - Adds a worker to a client's favourites or a client to a worker's favourites. Validates both the client and worker exist and checks that the relationship isn't already favourited before inserting the record. Used when a client bookmarks a worker or a worker bookmarks a client.

```

CREATE OR REPLACE PROCEDURE add_favourite(
    p_client_id INT,
    p_worker_id INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Verify client exists
    IF NOT EXISTS (SELECT 1 FROM Client WHERE id = p_client_id) THEN
        RAISE EXCEPTION 'Client % does not exist', p_client_id;
    END IF;

    -- Verify worker exists
    IF NOT EXISTS (SELECT 1 FROM Worker WHERE id = p_worker_id) THEN
        RAISE EXCEPTION 'Worker % does not exist', p_worker_id;
    END IF;

    -- Check if already favourited
    IF EXISTS (
        SELECT 1 FROM Favourite
        WHERE client_id = p_client_id AND worker_id = p_worker_id
    ) THEN
        RAISE EXCEPTION 'Worker % is already favourited by client %',
            p_worker_id, p_client_id;
    END IF;

    -- Create favourite

```

```

INSERT INTO Favourite (worker_id, client_id)
VALUES (p_worker_id, p_client_id);

RAISE NOTICE 'Worker % favourited by client %', p_worker_id, p_client_id;
END;
$$;

```

14. `remove_favourite` - Removes a worker from a client's favourites or a client from a worker's favourites. Validates the favourite relationship exists before deleting it. Used when a client unbookmarks a worker or a worker unbookmarks a client.

```

CREATE OR REPLACE PROCEDURE remove_favourite(
    p_client_id INT,
    p_worker_id INT
)
LANGUAGE plpgsql
AS $$
BEGIN
    -- Verify favourite exists
    IF NOT EXISTS (
        SELECT 1 FROM Favourite
        WHERE client_id = p_client_id AND worker_id = p_worker_id
    ) THEN
        RAISE EXCEPTION 'Worker % is not favourited by client %', p_worker_id,
p_client_id;
    END IF;

    -- Remove favourite
    DELETE FROM Favourite
    WHERE client_id = p_client_id AND worker_id = p_worker_id;

    RAISE NOTICE 'Worker % removed from client % favourites', p_worker_id,
p_client_id;
END;
$$;

```

Functions:

1. `clean_text` - Transliterates Cyrillic characters to their Latin equivalents using a character-by-character mapping. The function is declared IMMUTABLE, making it safe for use in index expressions and constant-folding optimizations by the query planner. Used when filtering a worker or client by name.

```

CREATE OR REPLACE FUNCTION clean_text(input TEXT)
RETURNS TEXT AS $$
BEGIN
    RETURN TRANSLATE(input,

```

```

        'абвгдґежзсијклљмнњопрсткѹфхцџшАБВГДґЕЖЗСИЈКЛЉМНЊОПРСТЌУФХЦЏШ',
        'abvgdgezzsijklmnpjoprstkuhfccdsABVGDGEZZSIJKLJMJNJOPRSTKUFHCCDS'
    );
END;
$$ LANGUAGE plpgsql IMMUTABLE;

```

2. `fn_match_workers` – Matches qualified workers to a specific task request based on multiple criteria including category expertise, work mode compatibility, geographic proximity, performance metrics, and current workload. Returns a ranked list of the best-matching workers. Used when a client is searching for workers for the task.

```

CREATE OR REPLACE FUNCTION fn_match_workers(
    p_task_request_id INT,
    p_limit           INT DEFAULT 20
)
RETURNS TABLE (
    out_worker_id      INT,
    out_user_id        INT,
    out_name            VARCHAR,
    out_surname         VARCHAR,
    out_city            VARCHAR,
    out_work_mode       VARCHAR,
    out_service_radius_km INT,
    out_distance_km     NUMERIC,
    out_categories      TEXT[],
    out_avg_rating      NUMERIC,
    out_total_reviews   BIGINT,
    out_performance_level VARCHAR,
    out_active_badge    VARCHAR,
    out_badge_tier      INT,
    out_active_tasks    BIGINT,
    out_match_score     NUMERIC
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_category_id      INT;
    v_parent_cat_id    INT;
    v_work_mode        VARCHAR(20);
    v_location_id      INT;
    v_task_lat         DECIMAL(9,6);
    v_task_lon         DECIMAL(9,6);
    v_task_created     TIMESTAMP;
BEGIN
    SELECT tr.category_id,
           tr.work_mode,
           tr.location_id,
           tr.created_at
    INTO v_category_id, v_work_mode, v_location_id, v_task_created
    FROM TaskRequest tr

```

```
WHERE tr.id = p_task_request_id;
```

```
IF NOT FOUND THEN
```

```
    RAISE EXCEPTION 'TaskRequest % does not exist', p_task_request_id;
```

```
END IF;
```

```
SELECT l.latitude, l.longitude
INTO   v_task_lat, v_task_lon
FROM   Location l
WHERE  l.id = v_location_id;
```

```
SELECT c.parent_category_id
INTO   v_parent_cat_id
FROM   Category c
WHERE  c.id = v_category_id;
```

```
RETURN QUERY
WITH
```

```
cat_workers AS (
    SELECT
        w.id                AS cw_worker_id,
        w.user_id           AS cw_user_id,
        w.work_mode         AS cw_work_mode,
        w.service_radius_km AS cw_radius,
        w.location_id       AS cw_location_id,
        w.created_at        AS cw_created
    FROM Worker w
    JOIN WorkerCategory wc ON wc.worker_id = w.id
    WHERE wc.category_id = v_category_id
    AND w.created_at < v_task_created
),
```

```
mode_workers AS (
    SELECT cw.*
    FROM   cat_workers cw
    WHERE  cw.cw_work_mode = v_work_mode
    OR     cw.cw_work_mode = 'HYBRID'
    OR     v_work_mode      = 'HYBRID'
),
```

```
location_workers AS (
    SELECT
        mw.cw_worker_id,
        mw.cw_user_id,
        mw.cw_work_mode,
        mw.cw_radius,
        l.city                AS lw_city,
        CASE
            WHEN v_work_mode = 'REMOTE' THEN NULL
            ELSE ROUND((
                6371 * 2 * ASIN(SQRT(
                    POWER(SIN(RADIANS(v_task_lat - l.latitude) / 2), 2)
                    + COS(RADIANS(l.latitude))
                    * COS(RADIANS(v_task_lat))
                    * POWER(SIN(RADIANS(v_task_lon - l.longitude) / 2), 2)
                ))
            )::NUMERIC, 2)
        AS lw_distance_km
    FROM   mode_workers mw
```

```

JOIN Location l ON l.id = mw.cw_location_id
WHERE v_work_mode = 'REMOTE'
OR (
    6371 * 2 * ASIN(SQRT(
        POWER(SIN(RADIANS(v_task_lat - l.latitude) / 2), 2)
        + COS(RADIANS(l.latitude))
        * COS(RADIANS(v_task_lat))
        * POWER(SIN(RADIANS(v_task_lon - l.longitude) / 2), 2)
    )) <= mw.cw_radius
)
),

worker_categories AS (
SELECT wc.worker_id AS wcat_worker_id,
    ARRAY_AGG(c.category_name::TEXT ORDER BY c.category_name) AS
wcat_categories
FROM WorkerCategory wc
JOIN Category c ON c.id = wc.category_id
WHERE wc.worker_id IN (
    SELECT w.id FROM Worker w
    JOIN WorkerCategory wc2 ON wc2.worker_id = w.id
    WHERE wc2.category_id = v_category_id
)
GROUP BY wc.worker_id
),

rating_stats AS (
SELECT
    r.reviewed_id AS rs_user_id,
    ROUND(AVG(r.rating), 2) AS rs_avg_rating,
    COUNT(r.id) AS rs_total_reviews
FROM Review r
GROUP BY r.reviewed_id
),

best_badge AS (
SELECT DISTINCT ON (wb.worker_id)
    wb.worker_id AS bb_worker_id,
    b.badge_name AS bb_badge_name,
    b.tier_level AS bb_tier_level
FROM WorkerBadge wb
JOIN Badge b ON b.id = wb.badge_id
WHERE wb.is_active = TRUE
    AND b.category_id = v_parent_cat_id
ORDER BY wb.worker_id, b.tier_level DESC
),

active_load AS (
SELECT o.worker_id AS al_worker_id, COUNT(t.id) AS al_active_tasks
FROM Task t
JOIN Offer o ON o.id = t.offer_id
WHERE t.status = 'ACTIVE'
    AND o.worker_id IN (
        SELECT w.id FROM Worker w
        JOIN WorkerCategory wc ON wc.worker_id = w.id
        WHERE wc.category_id = v_category_id
    )
GROUP BY o.worker_id
),

```



```

scored AS (
    SELECT
        lw.cw_worker_id AS sc_worker_id,
        lw.cw_user_id AS sc_user_id,
        u.name AS sc_name,
        u.surname AS sc_surname,
        lw.lw_city AS sc_city,
        lw.cw_work_mode AS sc_work_mode,
        lw.cw_radius AS sc_radius,
        lw.lw_distance_km AS sc_distance_km,
        COALESCE(wcat.wcat_categories, '{}') AS sc_categories,
        COALESCE(rs.rs_avg_rating, 0) AS sc_avg_rating,
        COALESCE(rs.rs_total_reviews, 0) AS sc_total_reviews,
        CASE
            WHEN COALESCE(rs.rs_avg_rating, 0) >= 4.5 THEN 'TOP'
            WHEN COALESCE(rs.rs_avg_rating, 0) >= 3.5 THEN 'MEDIUM'
            ELSE 'LOW'
        END AS sc_performance_level,
        bb.bb_badge_name AS sc_active_badge,
        bb.bb_tier_level AS sc_badge_tier,
        COALESCE(al.al_active_tasks, 0) AS sc_active_tasks,

        ROUND((
            COALESCE(rs.rs_avg_rating, 0) / 5.0 * 50.0

            + LEAST(LN(COALESCE(rs.rs_total_reviews, 0) + 1) * 2.5, 10.0)

            + COALESCE(bb.bb_tier_level, 0) * 3.0

            + CASE
                WHEN v_work_mode = 'REMOTE' OR lw.lw_distance_km IS NULL
                THEN 10.0
                WHEN lw.cw_radius = 0 THEN 0.0
                ELSE GREATEST(0.0,
                    (1.0 - lw.lw_distance_km / NULLIF(lw.cw_radius, 0)) *
                    10.0
                )
            END

            - LEAST(COALESCE(al.al_active_tasks, 0) * 3.0, 15.0)

        )::NUMERIC, 2) AS sc_match_score

    FROM location_workers lw
    JOIN UserAccount u ON u.id = lw.cw_user_id
    LEFT JOIN worker_categories wcat ON wcat.wcat_worker_id = lw.cw_worker_id
    LEFT JOIN rating_stats rs ON rs.rs_user_id = lw.cw_user_id
    LEFT JOIN best_badge bb ON bb.bb_worker_id = lw.cw_worker_id
    LEFT JOIN active_load al ON al.al_worker_id = lw.cw_worker_id
)

```

```

SELECT
    sc_worker_id,
    sc_user_id,
    sc_name,
    sc_surname,
    sc_city,
    sc_work_mode,
    sc_radius,
    sc_distance_km,
    sc_categories,
    sc_avg_rating,
    sc_total_reviews,
    sc_performance_level,
    sc_active_badge,
    sc_badge_tier,
    sc_active_tasks,
    sc_match_score
FROM scored
ORDER BY sc_match_score DESC
LIMIT p_limit;

```

```

END;
$$;

```

Triggers:

1. fn_set_updated_at / trg_*_updated_at - A shared trigger function that automatically sets the updated_at column to the current timestamp before any UPDATE on the registered tables. Applied to five tables via individual trigger definitions.

```

CREATE OR REPLACE FUNCTION fn_set_updated_at()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_worker_updated_at
BEFORE UPDATE ON Worker
FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_useraccount_updated_at
BEFORE UPDATE ON UserAccount
FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_taskrequest_updated_at
BEFORE UPDATE ON TaskRequest
FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_offer_updated_at
BEFORE UPDATE ON Offer

```

```

FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_payment_updated_at
BEFORE UPDATE ON Payment
FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

```

2. trg_notification_created_at - Enforces temporal integrity by ensuring that a Notification's created_at timestamp is strictly after the created_at timestamp of the associated UserAccount. Prevents back-dated notifications from being inserted or updated into the table.

```

CREATE OR REPLACE FUNCTION check_notification_created_at()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.created_at <= (SELECT created_at FROM UserAccount WHERE id =
NEW.user_id) THEN
        RAISE EXCEPTION 'notification created_at must be after user created_at';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_notification_created_at
BEFORE INSERT OR UPDATE ON Notification
FOR EACH ROW
EXECUTE FUNCTION check_notification_created_at();

```