

# **Project Report: Bakery Shop Manager Platform**

**Natthakul Yikusung 6680972**

## **Database access**

Hostname: dpg-d4jugo0gjchc739q8og0-a

Port: 5432

Database: remote\_pj\_db

Username: root

Password: Am5EJg98QKQARvkBhmLtsGDqervl8qoi

Internal Database URL: postgresql://root:Am5EJg98QKQARvkBhmLtsGDqervl8qoi@dpg-d4jugo0gjchc739q8og0-a/remote\_pj\_db

External Database URL: postgresql://root:Am5EJg98QKQARvkBhmLtsGDqervl8qoi@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com/remote\_pj\_db

PSQL Command: PGPASSWORD=Am5EJg98QKQARvkBhmLtsGDqervl8qoi psql -h dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com -U root remote\_pj\_db

## Table of Contents

<b>1. Introduction</b>	<b>3</b>
<b>2. Problem Statement (Pain Points)</b>	
<b>3. Solution: Platform Features and Value Proposition</b>	
<b>4. Database and Application Design</b>	<b>4</b>
<b>a. Entity-Relationship (ER) Diagram and All Tables Details</b>	<b>5</b>
<b>b. Application Flow (Sitemap)</b>	<b>14</b>
<b>c. Key Functionality Analysis</b>	<b>15</b>
<b>5. Financial Model</b>	
<b>6. Conclusion and Future Work</b>	
<b>a. Project Repository and Assets</b>	

## 1. Introduction

This report details the development of a comprehensive **Shop Management Platform** designed to streamline operations for small-scale businesses, specifically focusing on custom-order services. The platform centralizes key administrative tasks, including staff management, ingredient/inventory tracking, and customer interaction. The goal is to replace disparate, manual processes (like paper-based tracking) with a unified, digital system to enhance efficiency and profitability.

---

## 2. Problem Statement (Pain Points)

Small bakery owners face significant operational friction that consumes valuable time and can lead to financial loss. The platform directly addresses the following core pain points:

- **Wasting Time Answering Simple Questions:** Bakers frequently stop their production work to answer repetitive customer inquiries regarding availability.
  - **Getting Incomplete Orders:** Orders often lack necessary details (e.g., size, date, flavor), requiring lengthy follow-up communication and delaying the start of the order.
  - **Losing Track of Orders and Deadlines:** Relying on paper or simple spreadsheets makes it easy to forget deadlines or lose track of an order's current status (Pending, In Process, Finished).
  - **Hard Time Tracking Money and Costs:** Manual tracking of daily income, costs, and ingredient expenses prevents the baker from having a real-time view of profitability and ingredient cost.
  - **Lack of Final Confirmation:** Customers sometimes forget to confirm or pay the required deposit after receiving the final quote, which can lead to wasted effort and booking conflicts.
- 

## 3. Solution: Platform Features and Value Proposition

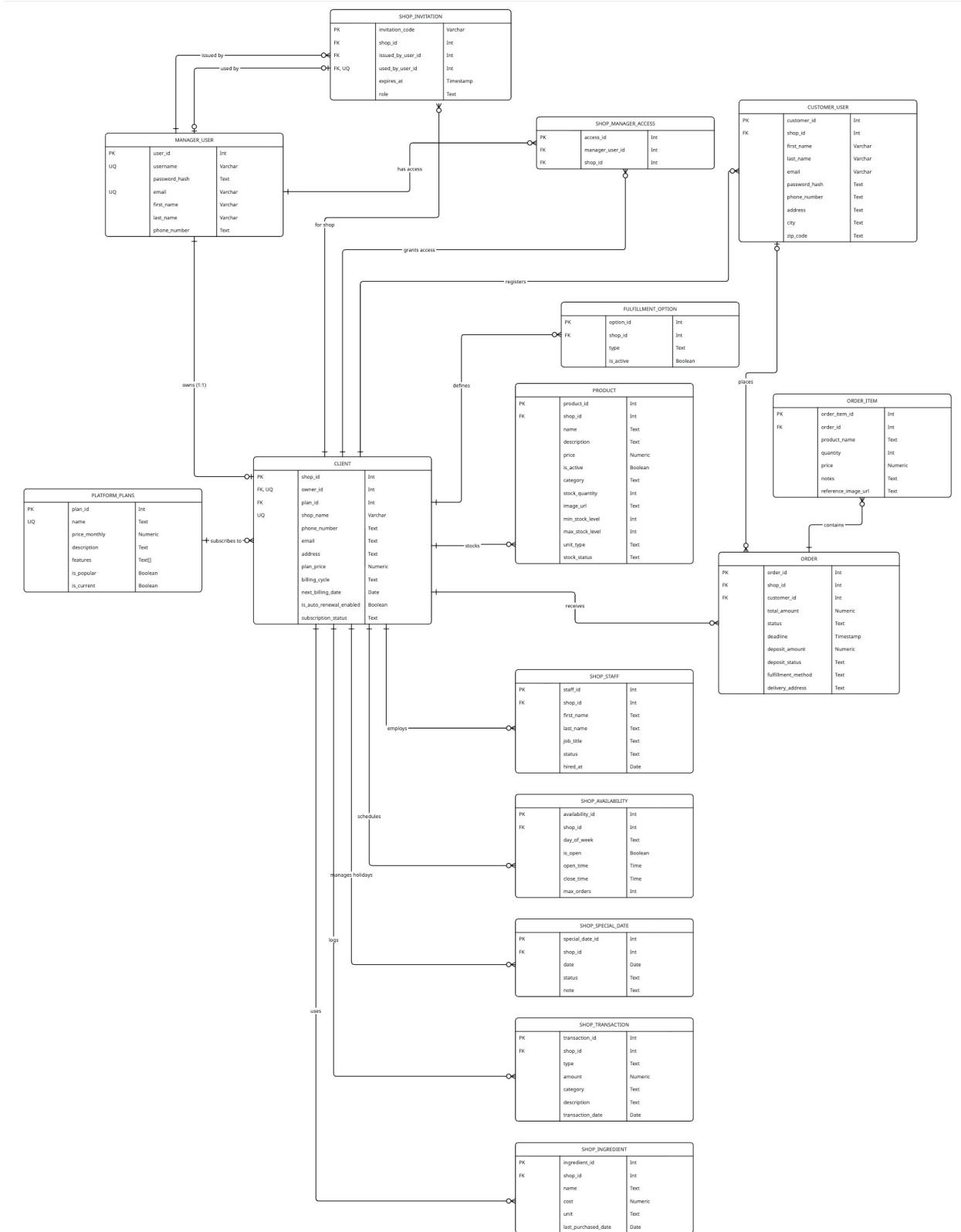
Pain Point	Platform Feature	Value Proposition
Wasting Time Answering Simple Questions	Automated Availability Schedule	Allows customers to self-service check for openings, freeing the baker to focus on production.
Getting Incomplete Orders	Detailed Customer Order Form	<b>Mandatory fields</b> ensure all necessary information (date, size, flavor, reference pictures) is collected in a single submission.

Losing Track of Orders and Deadlines	Order Tracking Dashboard with Reminders	Provides a clear, at-a-glance view of all orders and their status, with automated reminders to prevent missed deadlines.
Hard Time Tracking Money and Costs	Digital Income and Cost Tracker	Enables digital logging of all financial transactions for easy access to <b>monthly profits</b> and detailed <b>ingredient cost tracking</b> .
Lack of Final Confirmation	Automated Confirmation & Deposit Reminder	After a final price is sent, the system automatically sends a link for payment and confirmation, with a simple reminder after 24 hours if no response is received.

## 4. Database and Application Design

### Entity-Relationship (ER) Diagram and All Table Details

The system's database structure is critical for maintaining data integrity and relationships between core entities like shops, managers, staff, and ingredients.



## manager\_user

**Purpose:** Stores all platform-level user accounts (managers, owners, admins). These are the users who manage shops, staff accounts, customers, etc.

### Important Columns

Column	Description
user_id	Primary key, unique identifier for each manager user.
username	Unique login username.
password_hash	Secure hashed password for authentication.
email	Unique email address.
first_name, last_name	Optional personal information.
phone_number	Optional contact number.

### Keys & Constraints

- **Primary Key:** user\_id  
Ensures each user is uniquely identifiable.
- **Unique Keys:**
  - username (no two users can share username)
  - email (no duplicated emails allowed)
- These constraints prevent account duplication and ensure login integrity.

	user_id	username	password_hash	email	first_name	last_name	phone_number
1	28	admin1	hash_admin1	admin1@example.com	Alice	Smith	555-1001
2	29	manager2	hash_manager2	manager2@example.com	Bob	Johnson	555-1002
3	30	staff3	hash_staff3	staff3@example.com	Charlie	Brown	555-1003

## platform\_plans

**Purpose:** Holds the subscription plans available on the platform (e.g., Basic, Standard, Premium).

### Important Columns

- name: Unique plan name
- price\_monthly: Cost of the plan
- features: Array of text listing available features
- is\_popular, is\_current: Flags for display or filtering purposes

### Keys

- **Primary Key:** plan\_id

- **Unique Constraint:** name

Ensures no two plans share the same name.

plan_id	name	price_monthly	description	features	is_popular	is_current
1	16 Basic	19.99	Starter plan	{10 products, Basic support}	• true	false
2	17 Standard	29.99	Standard plan	{50 products, Priority support}	• true	• true
3	18 Premium	49.99	Premium plan	{Unlimited products, 24/7 support}	false	false

## client

**Purpose:** Represents registered shops/businesses using the platform.

### Key Columns

Column	Description
shop_id	Unique shop identifier
owner_id	FK → manager_user.user_id, the account owning the shop
shop_name	Unique name of the shop
plan_id	FK → platform_plans.plan_id to determine subscription
Various billing columns	Handle subscription management

### Keys

- **Primary Key:** shop\_id
- **Foreign Keys:**
  - owner\_id → manager\_user
  - plan\_id → platform\_plans
- **Unique:** owner\_id (one owner = one shop), shop\_name

This ensures ownership integrity.

shop_id	owner_id	shop_name	phone_number	email	address	plan_price	billing_cycle	next_billing_date
1	20	28 Alice Boutique	555-2001	alice_shop@example.com	123 Market Street	29.99	Monthly	2025-12-29
2	21	29 Bob Electronics	555-2002	bob_elec@example.com	456 Tech Road	49.99	Monthly	2025-12-29

is_auto_renewal_enabled	subscription_status	plan_id
• true	Active	17
• true	Active	18

## shop\_manager\_access

**Purpose:** Allows multiple managers to access a shop. This supports roles such as “staff”, “assistant manager”.

### Important Columns

- manager\_user\_id → manager\_user

- shop\_id → client

## Keys

- **Primary Key:** access\_id
- **Foreign Keys:**
  - manager\_user\_id
  - shop\_id
- **Unique Composite:** (manager\_user\_id, shop\_id)  
Prevents duplicate assignments of the same manager to the same shop.

	access_id	manager_user_id	shop_id
1		24	28
2		25	29
3		26	30

## customer\_user

**Purpose:** Stores customers belonging to each shop (not platform-level users).

### Important Columns

- customer\_id: Primary key
- shop\_id: FK to identify which shop owns the customer
- Customer profile fields: names, email, password

## Keys

- **Primary Key:** customer\_id
- **Foreign Key:** shop\_id → client
- **Unique Constraints:**
  - (shop\_id, email)
  - (shop\_id, phone\_number)

This means **customers are unique within a shop**, but different shops may share customers.

customer_id	shop_id	first_name	last_name	email	password_hash	phone_number	address	city	zip_code	created_at
1	20	John	Doe	john.doe@example.com	hash_johndoe	555-3001	12 Elm St	Springfield	12345	2025-11-29 21:41:49.944861 +00:00
2	21	Jane	Roe	jane.roe@example.com	hash_janeroe	555-3002	34 Pine St	Springfield	12345	2025-11-29 21:41:49.944861 +00:00
3	22	Tom	Black	tom.black@example.com	hash_tomblack	555-3003	89 Oak St	Riverside	67898	2025-11-29 21:41:49.944861 +00:00

## shop\_invitation

**Purpose:** Manages invitations to shops (for adding staff/managers).

### Important Columns

- invitation\_code: Primary key, unique code.
- shop\_id: FK → client
- issued\_by\_user\_id: FK → manager\_user
- used\_by\_user\_id: FK → manager\_user (nullable)

### Keys

- **Primary Key:** invitation\_code
- **Unique:** used\_by\_user\_id  
Ensures one invitation can be used by only one user.

	invitation_code	shop_id	issued_by_user_id	expires_at	used_by_user_id	created_at	email	role
1	1234567890ABCDE1234567890ABC	20		28 2025-12-06 21:43:20.993439		28 2025-11-29 21:43:20.993439	staff3@example.com	Staff
2	0987654321XYZ0987654321XYZ	21		29 2025-12-06 21:43:20.993439	<null>	2025-11-29 21:43:20.993439	newhire@example.com	Staff

### fulfillment\_option

**Purpose:** Defines available order fulfillment methods per shop (e.g., Pickup, Delivery).

### Important Columns

- shop\_id: FK → client
- type: e.g., "Pick Up" or "Delivery"

### Keys

- **Primary Key:** option\_id
- **Unique constraint:** (shop\_id, type)  
Prevents duplicates per shop.

	option_id	shop_id	type	is_active
1		11	20 Pick Up	• true
2		12	20 Delivery	• true
3		13	21 Pick Up	• true

### product

**Purpose:** Stores products for each shop.

### Important Columns

- shop\_id: FK → client

- name, description, price
- Stock-related fields (stock\_quantity, min\_stock\_level, etc.)
- stock\_status: Computed column (Generated Always)

## Keys

- **Primary Key:** product\_id

**Generated Column Explanation:** The stock\_status field automatically evaluates to:

- “Low Stock”
- “Overstocked”
- “Medium”

based on quantity thresholds. This ensures consistent inventory evaluation logic.

product_id	shop_id	name	description	price	is_active	created_at	category	stock_quantity
1	16	20 Handmade Candle	Lavender scented candle	15.99	true	2025-11-29 21:43:56.052841 +00:00	Home Decor	20
2	17	21 Wireless Mouse	Ergonomic mouse	25.50	true	2025-11-29 21:43:56.052841 +00:00	Electronics	80
image_url	updated_at	min_stock_level	max_stock_level	unit_type	last_restocked_at	stock_status		
https://example.com/candle.jpg	2025-11-29 21:43:56.052841 +00:00	5	50	pieces	2025-11-26 21:43:56.052841 +00:00	Medium		
https://example.com/mouse.jpg	2025-11-29 21:43:56.052841 +00:00	10	100	pieces	2025-11-24 21:43:56.052841 +00:00	Medium		

## order

**Purpose:** Represents customer orders.

### Important Columns

- shop\_id: FK → client
- customer\_id: optional FK
- status: Restricted by a CHECK constraint to valid statuses
- fulfillment\_method: CHECK constraint for ‘Pick Up’ or ‘Delivery’

## Keys

- **Primary Key:** order\_id
- **Foreign Key:** shop\_id

The constraints guarantee valid workflow transitions.

order_id	shop_id	customer_id	total_amount	status	created_at	deadline
1	40	20	20	45.99	Confirmed	2025-11-29 21:45:39.442277 +00:00
2	41	21	22	25.50	Awaiting Confirmation	2025-11-29 21:45:39.442277 +00:00
deposit_amount	deposit_status	notes	fulfillment_method	delivery_address	payment_slip_url	
10.00	Paid	Please deliver ASAP	Delivery	12 Elm St	https://example.com/slip1.jpg	
0.00	Pending	Pick up after 5 PM	Pick Up	<null>	<null>	

## shop\_staff

**Purpose:** Stores staff employees of each shop.

### Important Columns

- shop\_id: FK
- first\_name, last\_name
- email, phone\_number

### Keys

- **Primary Key:** staff\_id

Ensures staff records are tied to the correct shop.

staff_id	shop_id	first_name	last_name	email	phone_number	job_title	status	hired_at
1	12	Emily	Stone	emily.staff@example.com	555-4001	Cashier	active	2025-11-29
2	13	Frank	Miller	frank.staff@example.com	555-4002	Technician	active	2025-11-29

## shop\_availability

**Purpose:** Defines business days and working hours for each shop.

### Key Columns

- shop\_id
- day\_of\_week
- open\_time, close\_time

### Keys

- **Primary Key:** availability\_id
  - **Unique Composite:** (shop\_id, day\_of\_week)
- Each shop can have *only one schedule entry per day*.

availability_id	shop_id	day_of_week	is_open	open_time	close_time	max_orders	created_at	updated_at
1	11	Monday	true	09:00:00	18:00:00	20	2025-11-29 21:46:32.556936 +00:00	2025-11-29 21:46:32.556936 +00:00
2	12	Tuesday	true	09:00:00	18:00:00	20	2025-11-29 21:46:32.556936 +00:00	2025-11-29 21:46:32.556936 +00:00
3	13	Monday	true	10:00:00	19:00:00	15	2025-11-29 21:46:32.556936 +00:00	2025-11-29 21:46:32.556936 +00:00

## shop\_special\_date

**Purpose:** Manages exceptions, such as holidays or special events.

### Important Columns

- shop\_id
- date

- status: e.g., “Closed”, “Special Event”

## Keys

- **Primary Key:** special\_date\_id
- **Unique Composite:** (shop\_id, date)  
Prevents two rules for the same day.

special_date_id	shop_id	date	status	note	created_at	updated_at
1	9	2025-12-04	Closed	Holiday	2025-11-29 21:46:43.099539 +00:00	2025-11-29 21:46:43.099539 +00:00
2	10	2025-12-09	Open	Special sale	2025-11-29 21:46:43.099539 +00:00	2025-11-29 21:46:43.099539 +00:00

## order\_item

**Purpose:** Stores items inside an order.

### Important Columns

- order\_id: FK → order
- product\_name
- quantity, price

## Keys

- **Primary Key:** order\_item\_id
- **Foreign Key:** order\_id

order_item_id	order_id	product_name	quantity	price	notes	created_at	reference_image_url
1	16	40 Handmade Candle	2	15.99	Gift wrap	2025-11-29 21:47:01.846498 +00:00	https://example.com/item1.jpg
2	17	41 Wireless Mouse	1	25.50	Test before pickup	2025-11-29 21:47:01.846498 +00:00	https://example.com/item2.jpg

## shop\_transaction

**Purpose:** Tracks financial transactions per shop (income or expenses).

### Important Columns

- shop\_id
- type (Income / Expense)
- amount, category

## Keys

- **Primary Key:** transaction\_id
- **Foreign Key:** shop\_id
- **CHECK constraint** ensures valid transaction types.

## shop\_ingredient

**Purpose:** Stores materials/ingredients used by shops (e.g., candle shops, bakeries, etc.).

### Important Columns

- shop\_id
- name
- cost
- unit

### Keys

- **Primary Key:** ingredient\_id
- **Foreign Key:** shop\_id

Used for inventory and cost tracking.

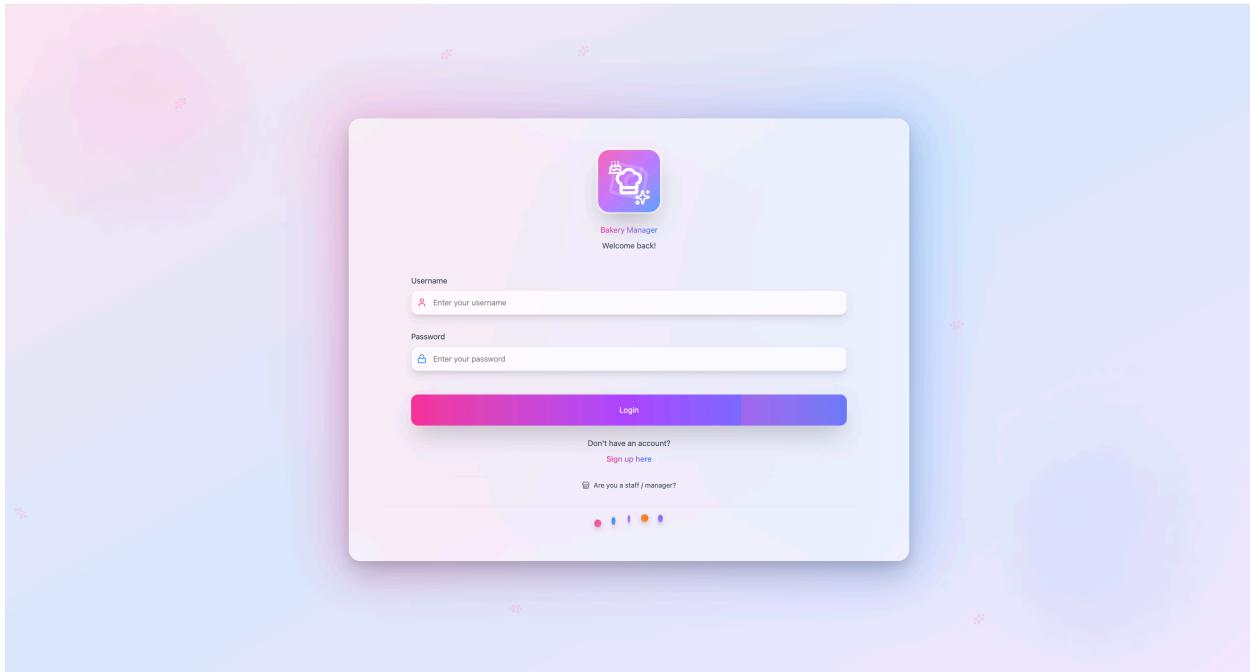
ingredient_id	shop_id	name	cost	unit	last_purchased_date	created_at
1	13	28 Wax	10.50	kg	2025-11-19	2025-11-29 21:47:26.900720 +00:00
2	14	28 Lavender Oil	5.25	bottle	2025-11-22	2025-11-29 21:47:26.900720 +00:00
3	15	21 Circuit Board	15.00	piece	2025-11-14	2025-11-29 21:47:26.900720 +00:00

## Application Flow (Sitemap)

The sitemap illustrates the user journey and navigational hierarchy within the platform, showing how users interact with modules like the Shop Profile and Staff Members.

## Key Functionality Analysis

### a. User Management and Security



#### `fn_login_unified`

- **Purpose:** Handles secure user authentication, differentiating between platform managers and shop customers based on credentials and the target shop.
- **How It Works (Logic):** First, it verifies the `p_shop_identifier` to retrieve the necessary `shop_id`. If found, it attempts two distinct login checks:
  - 1) **Manager Login**, joining `manager_user` with `shop_manager_access`, checking password hash securely with `crypt()`, and verifying access to the target shop.
  - 2) If manager login fails, it attempts **Customer Login** using `customer_user`, again using `crypt()` for the password check within the target shop's scope.

- **Code:**

```
create function fn_login_unified(p_username_or_email text, p_password text,
p_shop_identifier text)
  returns TABLE(auth_id integer, user_type text, shop_id integer, first_name
text)
  language plpgsql
as
$$
DECLARE
  v_target_shop_id INT;
BEGIN
```

```

SELECT client.shop_id INTO v_target_shop_id
FROM client
WHERE shop_name = p_shop_identifier
LIMIT 1;

IF v_target_shop_id IS NULL THEN
    RETURN;
END IF;

RETURN QUERY
SELECT
    mu.user_id AS auth_id,
    'manager'::TEXT AS user_type,
    sma.shop_id,
    mu.first_name::TEXT
FROM manager_user mu
    JOIN shop_manager_access sma ON mu.user_id = sma.manager_user_id
WHERE (mu.username = p_username_or_email OR mu.email =
p_username_or_email)
    AND mu.password_hash = crypt(p_password, mu.password_hash)
    AND sma.shop_id = v_target_shop_id
LIMIT 1;

IF FOUND THEN
    RETURN;
END IF;

RETURN QUERY
SELECT
    cu.customer_id AS auth_id,
    'customer'::TEXT AS user_type,
    cu.shop_id,
    cu.first_name::TEXT
FROM customer_user cu
WHERE cu.email = p_username_or_email
    AND cu.password_hash = crypt(p_password, cu.password_hash)
    AND cu.shop_id = v_target_shop_id
LIMIT 1;

RETURN;
END;
$$;

alter function fn_login_unified(text, text, text) owner to root;

```

- **Result Structure:** Returns the authorized user's details: 0-(auth\_id INTEGER, user\_type TEXT, shop\_id INTEGER, first\_name TEXT).

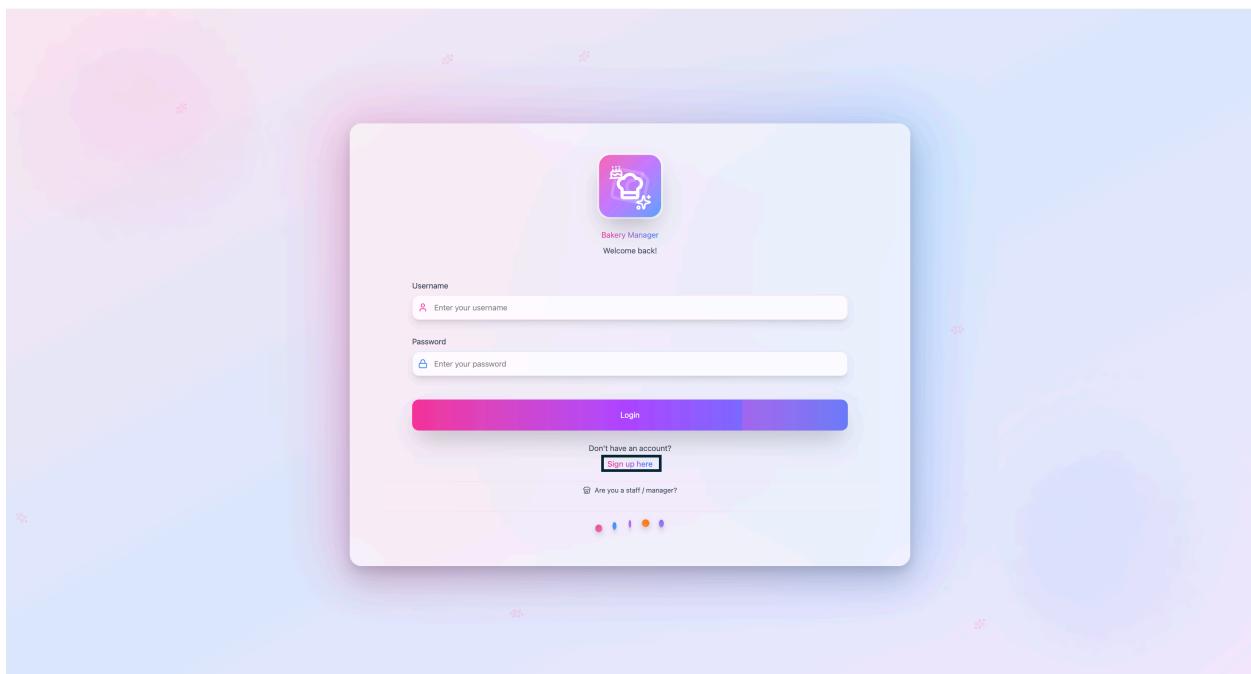
```
1 ✓ SELECT * FROM fn_login_unified( p_username_or_email 'new@customer.com', p_password 'Password1', p_shop_identifier 'Sweet Cake Shop');  
2
```

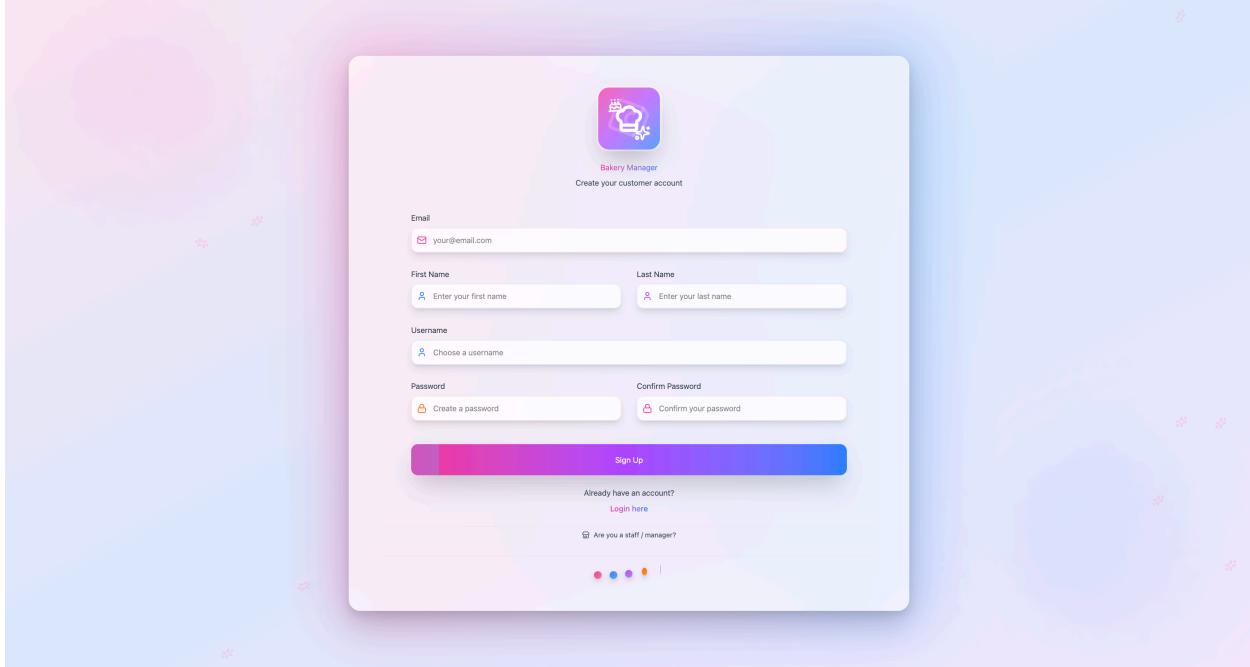
Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12

Tx Output Result 67 ×

auth\_id user\_type shop\_id first\_name

	auth_id	user_type	shop_id	first_name
1	23	customer		20 Jane





## fn\_customer\_register

- **Purpose:** Registers a new customer and securely links them to a specific shop.
- **How It Works (Logic):** Finds the target shop\_id using the shop identifier. It then inserts the customer data into customer\_user, securing the password using crypt(). Includes uniqueViolation exception handling to gracefully manage duplicate email/phone attempts within the same shop.
- **Code:**

```
create function fn_customer_register(p_shop_identifier text, p_email text,
p_password text, p_first_name text, p_last_name text, p_phone_number text,
p_address text DEFAULT NULL::text, p_city text DEFAULT NULL::text, p_zip_code text
DEFAULT NULL::text) returns integer
language plpgsql
as
$$
DECLARE
    v_shop_id INT;
    v_new_customer_id INT;
BEGIN
    -- 1. Get the shop ID based on the identifier
    SELECT shop_id INTO v_shop_id
    FROM client
    WHERE shop_name = p_shop_identifier
    LIMIT 1;

    IF v_shop_id IS NULL THEN
        RAISE EXCEPTION 'Shop not found: %', p_shop_identifier;
    END IF;
    -- 2. Insert the new customer
    INSERT INTO customer_user (shop_id, email, password, first_name, last_name, phone_number, address, city, zip_code)
    VALUES (v_shop_id, p_email, crypt(p_password, gen_salt('bf')), p_first_name, p_last_name, p_phone_number, p_address, p_city, p_zip_code);
    RETURN v_new_customer_id;
END;
```

```

END IF;

-- 2. Create the customer record (Password is now ENCRYPTED)
INSERT INTO customer_user (
    shop_id,
    first_name,
    last_name,
    email,
    password_hash,
    phone_number,
    address,
    city,
    zip_code
)
VALUES (
    v_shop_id,
    p_first_name,
    p_last_name,
    p_email,
    crypt(p_password, gen_salt('bf')), -- <--- ENCRYPTION HERE
    p_phone_number,
    p_address,
    p_city,
    p_zip_code
)
RETURNING customer_id INTO v_new_customer_id;

RETURN v_new_customer_id;

EXCEPTION
    WHEN uniqueViolation THEN
        RAISE EXCEPTION 'Registration failed: An account with this email already
exists for this shop.';
    END;
$$;

alter function fn_customer_register(text, text, text, text, text, text, text,
text, text) owner to root;

```

- **Result Structure:** Returns the ID of the newly created customer: INTEGER.

```

1 ✓ SELECT fn_customer_register(
2     p_shop_identifier 'Sweet Cake Shop', p_email 'new@customer.com', p_password 'Password1',
3     p_first_name 'Jane', p_last_name 'Doe', p_phone_number '099922333', p_address '123 Street', p_city 'City', p_zip_code '5000'
4 );
5

```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12

Tx Output fn\_customer\_register...000' ):integer ×

fn\_customer\_register ↴ :

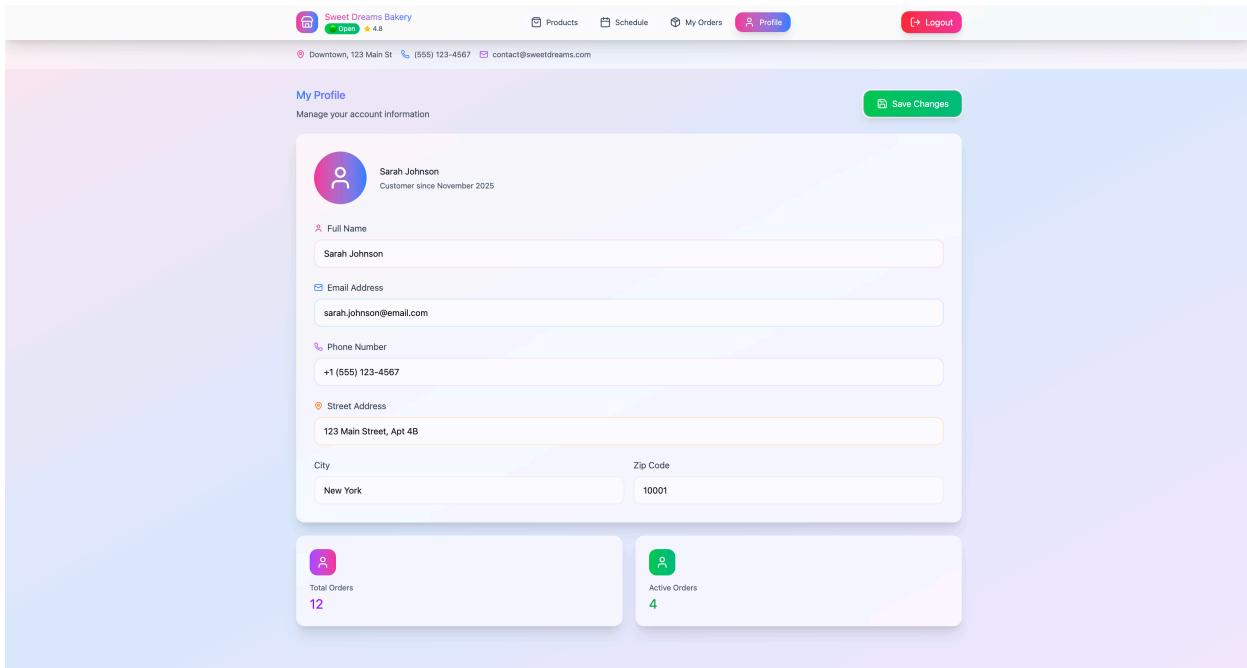
1	23				
customer_id	shop_id	first_name	last_name	email	password_hash
4	23	20 Jane	Doe	new@customer.com	\$2a\$06\$14P6S.mTuK27WB7Py590c08UbwsY./vKENF1kWn0xX3o0zrAmXM

The screenshot shows a web application interface for a bakery named "Sweet Dreams Bakery". At the top, there is a navigation bar with links for "Products", "Schedule", "My Orders", "Profile", and "Logout". Below the navigation, there is a header with the shop name, address ("Downtown, 123 Main St"), phone number ("(555) 123-4567"), email ("contact@sweatdreams.com"), and a rating of "4.8".

The main content area is titled "My Profile" and contains a sub-header "Manage your account information". It features a circular profile picture placeholder with a "J" icon, the name "Sarah Johnson", and the text "Customer since November 2025". Below this, there are input fields for "Full Name" (Sarah Johnson), "Email Address" (sarah.johnson@email.com), "Phone Number" (+1 (555) 123-4567), and "Street Address" (123 Main Street, Apt 4B). There are also separate input fields for "City" (New York) and "Zip Code" (10001).

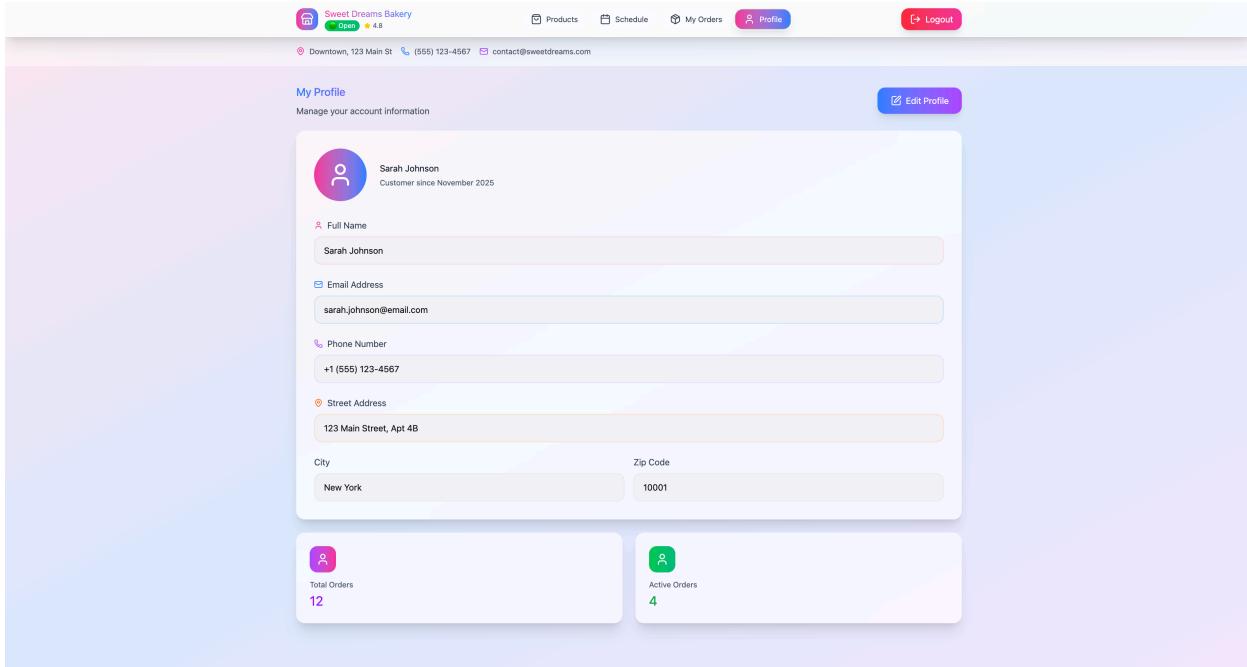
On the right side of the profile section, there is a blue "Edit Profile" button.

At the bottom of the profile section, there are two cards: "Total Orders" (12) and "Active Orders" (4).



## **fn\_update\_customer\_profile**

- **Purpose:** Allows an authenticated customer to modify their personal details.
  - **How It Works (Logic):** Executes a straightforward UPDATE query on the customer\_user table based on the primary key (customer\_id).
  - **Code:**
  - **Result Structure:** Returns BOOLEAN (True if the update was successful).



## fn\_get\_customer\_profile

- Purpose:** Retrieves a customer's personal details and relevant order statistics for their dashboard.
- How It Works (Logic):** Selects core data from customer\_user. It executes two nested subqueries against the order table to calculate total\_orders and active\_orders (excluding 'Finished' or 'Cancelled' status).
- Code:**

```
create function fn_get_customer_profile(p_customer_id integer)
    returns TABLE(first_name text, last_name text, email text, phone_number text,
    street_address text, city text, zip_code text, customer_since date, total_orders
    bigint, active_orders bigint)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        cu.first_name::TEXT,
        cu.last_name::TEXT,
        cu.email::TEXT,
        cu.phone_number::TEXT,
        cu.address::TEXT AS street_address,
        cu.city::TEXT,
        cu.zip_code::TEXT,
        cu.created_at::DATE AS customer_since,
```

```

-- Calculate Total Orders
(SELECT COUNT(*) FROM "order" o WHERE o.customer_id =
cu.customer_id)::BIGINT,

-- Calculate Active Orders (Not Finished/Canceled)
(SELECT COUNT(*) FROM "order" o
WHERE o.customer_id = cu.customer_id
AND o.status NOT IN ('Finished', 'Cancelled'))::BIGINT
FROM
customer_user cu
WHERE
cu.customer_id = p_customer_id;
END;
$$;

alter function fn_get_customer_profile(integer) owner to root;

```

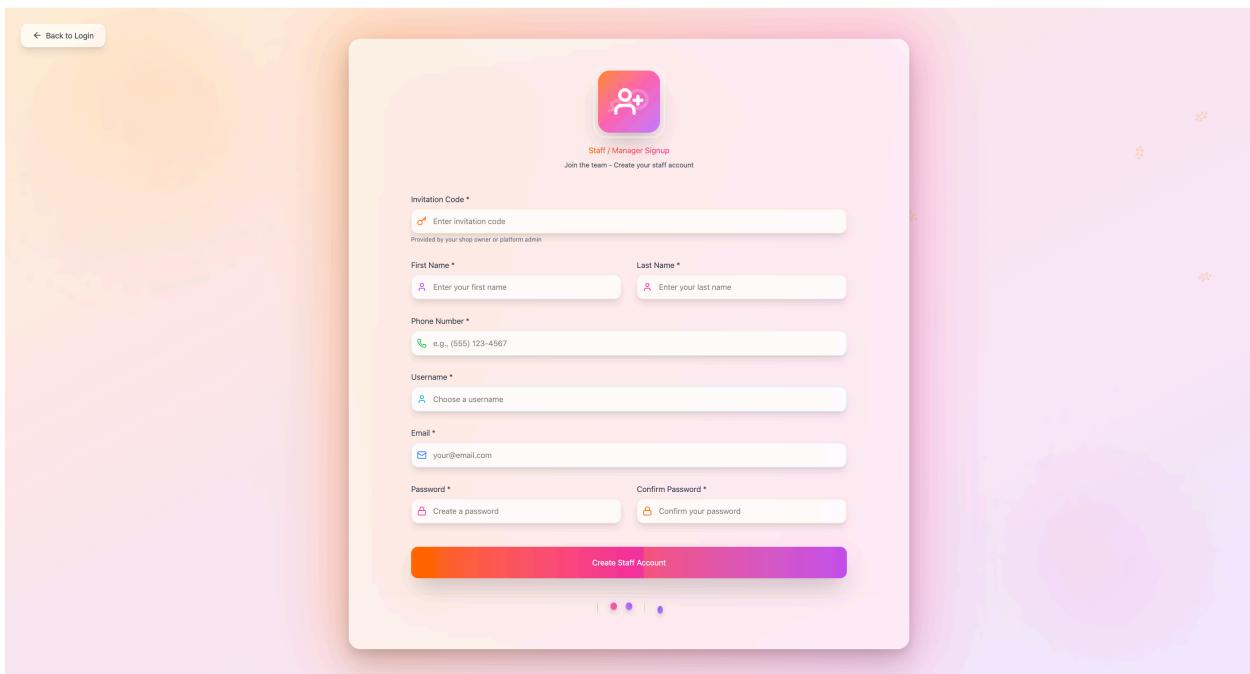
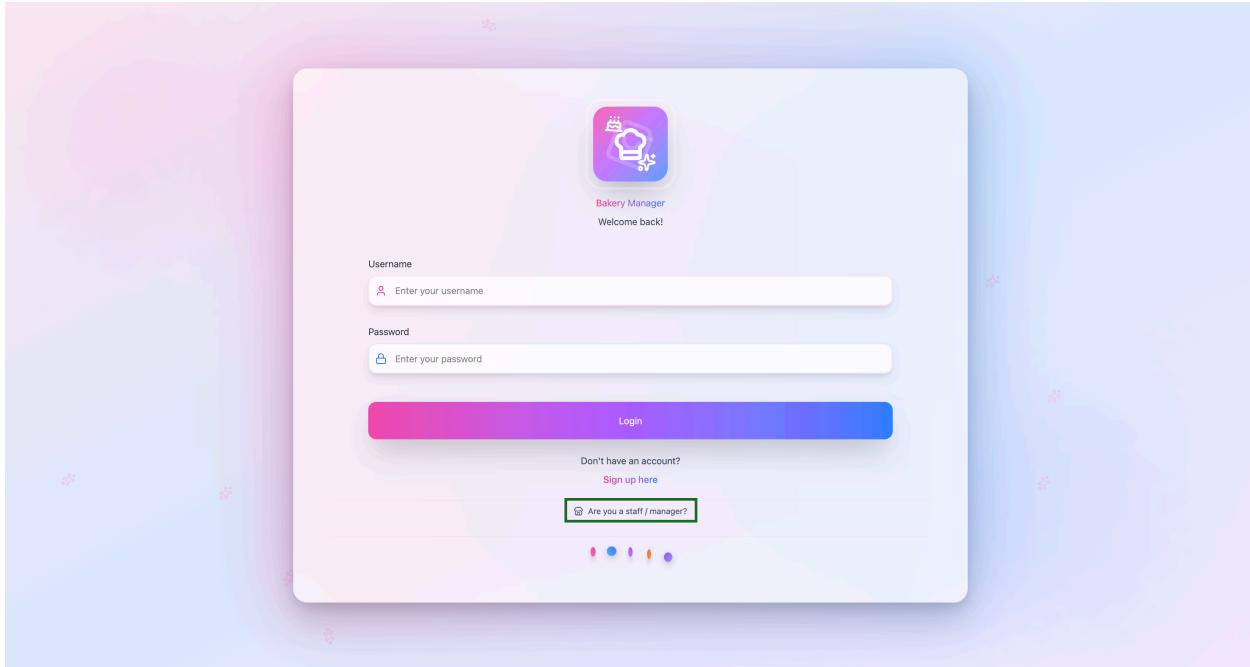
- Result Structure:** Returns comprehensive customer and order summary: (first\_name TEXT, last\_name TEXT, email TEXT, phone\_number TEXT, street\_address TEXT, city TEXT, zip\_code TEXT, customer\_since DATE, total\_orders BIGINT, active\_orders BIGINT).



The screenshot shows a PostgreSQL database interface. In the command line, the user has run the query: `SELECT * FROM fn_get_customer_profile(23);`. The result set is displayed in a table with the following data:

	first_name	last_name	email	phone_number	street_address	city	zip_code	customer_since	total_orders	active_orders
1	Jane	Doe	new@customer.com	099922333	123 Street	City	5000	2025-11-30	0	0

## b. Shop owner facing function



## fn\_register\_staff\_user

- Purpose:** Registers a new manager\_user (staff member) and grants them access to a shop using a secure invitation code, ensuring controlled staff onboarding.
- How It Works (Logic):** It begins with a critical validation step, checking the p\_invitation\_code validity and ensuring it hasn't been used yet (used\_by\_user\_id IS NULL) using FOR UPDATE to prevent concurrent use. Upon success, it inserts the new user into manager\_user, encrypting the password with crypt(p\_password, gen\_salt('bf')).

It then grants access via shop\_manager\_access and finally marks the invitation code as used.

- **Code:**

```

create function fn_register_staff_user(p_invitation_code text, p_username text,
p_email text, p_password text, p_first_name text, p_last_name text, p_phone_number
text) returns integer
    language plpgsql
as
$$
DECLARE
    v_new_user_id INT;
    v_shop_id INT;
BEGIN
    -- 1. Verify Invitation Code
    SELECT shop_id INTO v_shop_id
    FROM shop_invitation
    WHERE invitation_code = p_invitation_code
        AND used_by_user_id IS NULL
        FOR UPDATE;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Invalid or already used invitation code.';
    END IF;

    INSERT INTO manager_user (
        username,
        password_hash,
        email,
        first_name,
        last_name,
        phone_number
    )
    VALUES (
        p_username,
        crypt(p_password, gen_salt('bf')), -- <--- ENCRYPTION HERE
        p_email,
        p_first_name,
        p_last_name,
        p_phone_number
    )
    RETURNING user_id INTO v_new_user_id;

    INSERT INTO shop_manager_access (manager_user_id, shop_id)
    VALUES (v_new_user_id, v_shop_id);

    UPDATE shop_invitation
    SET used_by_user_id = v_new_user_id
    WHERE invitation_code = p_invitation_code;

```

```

    RETURN v_new_user_id;

EXCEPTION
    WHEN uniqueViolation THEN
        RAISE EXCEPTION 'Registration failed: Username or email already in use.';
END;
$$;

alter function fn_register_staff_user(text, text, text, text, text, text, text)
owner to root;

```

- **Result Structure:** Returns the ID of the newly created manager: INTEGER.

```

1 ✓   SELECT fn_register_staff_user(
2           p_invitation_code 'INVITE-NEW123',
3           p_username 'john.staff',
4           p_email 'newstaff@shop.com',
5           p_password 'Password123',
6           p_first_name 'John',
7           p_last_name 'Staff',
8           p_phone_number '09098887777'
9       );

```

The screenshot shows a PostgreSQL query editor with the following details:

- Query:** A SELECT statement calling the `fn_register_staff_user` function with several parameters. The parameters include an invitation code ('INVITE-NEW123'), a username ('john.staff'), an email ('newstaff@shop.com'), a password ('Password123'), first name ('John'), last name ('Staff'), and a phone number ('09098887777').
- Output:** The result of the query is displayed in a table. The table has two columns: a row number (1) and the result value (35). The table title is `fn_register_staff_us...777' )::integer`.

### fn\_validate\_invitation\_code

- **Purpose:** Checks the validity of an invitation code submitted by a potential new staff member.
- **How It Works (Logic):** Joins `shop_invitation` with `client` to retrieve the shop name, ensuring the code matches and that the `used_by_user_id` field remains NULL.
- **Code:**

```

create function fn_validate_invitation_code(p_invitation_code text)
    returns TABLE(is_valid boolean, shop_id integer, shop_name text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            TRUE AS is_valid,
            si.shop_id,
            c.shop_name
        FROM

```

```

shop_invitation si
    JOIN
    client c ON si.shop_id = c.shop_id
WHERE
    si.invitation_code = p_invitation_code
    AND si.used_by_user_id IS NULL;

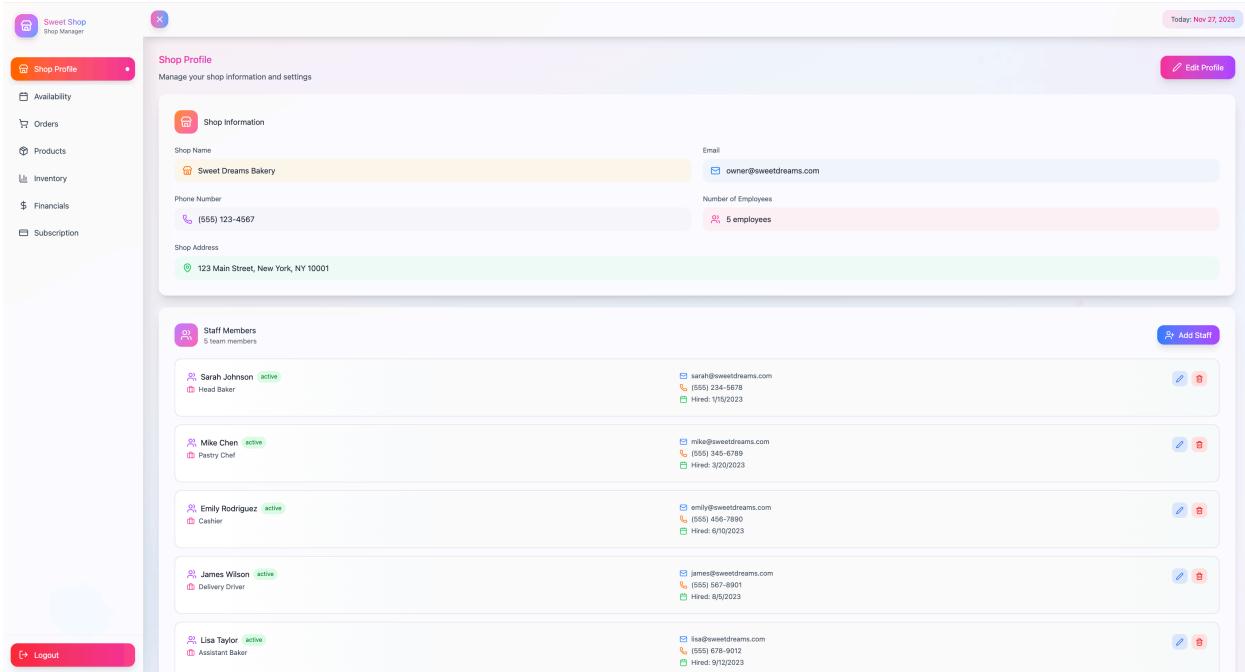
END;
$$;

alter function fn_validate_invitation_code(text) owner to root;

```

- **Result Structure:** Returns validation status and shop context: (is\_valid BOOLEAN, shop\_id INTEGER, shop\_name TEXT).

## Shop Management and Configuration



The screenshot shows the Sweet Shop Shop Manager interface. On the left, there's a sidebar with navigation links: Shop Profile (highlighted in pink), Availability, Orders, Products, Inventory, Financials, and Subscription. The main area is titled "Shop Profile" and says "Manage your shop information and settings". It contains fields for Shop Name (Sweet Dreams Bakery), Email (owner@sweetdreams.com), Phone Number ((655) 123-4567), and Number of Employees (5 employees). Below that is a "Shop Address" field with the value 123 Main Street, New York, NY 10001. To the right, there's a "Staff Members" section with a table listing five team members: Sarah Johnson (active, Head Baker), Mike Chen (active, Pastry Chef), Emily Rodriguez (active, Cashier), James Wilson (active, Delivery Driver), and Lisa Taylor (active, Assistant Baker). Each staff member has an email, phone number, and hire date listed next to their profile picture. There are edit and delete icons for each staff member.

### fn\_get\_shop\_profile\_info

- **Purpose:** Retrieves the shop's public profile and contact information for the admin dashboard.

- **How It Works (Logic):** Selects key fields from client. A separate subquery against shop\_manager\_access counts the distinct manager\_user\_ids associated with the shop to determine the platform team size (employee\_count).

- **Code:**

```

create function fn_get_shop_profile_info(p_shop_id integer)
    returns TABLE(shop_name text, email text, phone_number text, shop_address
text, employee_count bigint)
    language plpgsql
as
$$
DECLARE
    v_employee_count BIGINT;
BEGIN
    SELECT COUNT(DISTINCT sma.manager_user_id) INTO v_employee_count
    FROM shop_manager_access sma
    WHERE sma.shop_id = p_shop_id;

    RETURN QUERY
        SELECT
            c.shop_name::TEXT,
            c.email::TEXT,
            c.phone_number::TEXT,
            c.address::TEXT AS shop_address,
            v_employee_count
        FROM
            client c
        WHERE
            c.shop_id = p_shop_id;
END;
$$;

alter function fn_get_shop_profile_info(integer) owner to root;

```

- **Result Structure:** Returns shop and management information: (shop\_name TEXT, email TEXT, phone\_number TEXT, shop\_address TEXT, employee\_count BIGINT).

1 ✓ SELECT \* FROM fn\_get\_shop\_profile\_info( p\_shop\_id 20);

shop_name	email	phone_number	shop_address	employee_count
Alice Boutique	alice_shop@example.com	555-2001	123 Market Street	3

The application interface consists of two main sections. The left sidebar contains navigation links: Shop Profile (highlighted in pink), Availability, Orders, Products, Inventory, Financials, and Subscription. The right main area has a header with the date 'Today: Nov 27, 2025' and an 'Edit Profile' button.

**Shop Profile**

Manage your shop information and settings

**Shop Information**

Shop Name: Sweet Dreams Bakery | Email: owner@sweetdreams.com | Phone Number: (555) 123-4567 | Number of Employees: 5 employees

Shop Address: 123 Main Street, New York, NY 10001

**Staff Members** 5 team members

- Sarah Johnson (active) Head Baker | Email: sarah@sweetdreams.com | Phone: (555) 123-5678 | Hired: 7/1/2023
- Mike Chen (active) Pastry Chef | Email: mike@sweetdreams.com | Phone: (555) 345-6789 | Hired: 3/20/2023
- Emily Rodriguez (active) Cashier | Email: emily@sweetdreams.com | Phone: (555) 456-7890 | Hired: 9/10/2023
- James Wilson (active) Delivery Driver | Email: james@sweetdreams.com | Phone: (555) 667-8901 | Hired: 8/9/2023
- Lisa Taylor (active) Assistant Baker | Email: lisa@sweetdreams.com | Phone: (555) 678-9012 | Hired: 9/12/2023

**Logout**

**Shop Profile**

Manage your shop information and settings

**Shop Information**

Shop Name: Sweet Dreams Bakery | Email: owner@sweetdreams.com | Phone Number: (555) 123-4567 | Number of Employees: 5 employees

Shop Address: 123 Main Street, New York, NY 10001

**Save Changes**

## fn\_update\_shop\_profile\_info

- Purpose:** Allows a shop owner to update the shop's public contact information and address.
- How It Works (Logic):** Executes a straightforward UPDATE query on the client table based on the shop\_id.
- Code:**

```
create function fn_update_shop_profile_info(p_shop_id integer, p_shop_name text,
p_email text, p_phone_number text, p_shop_address text) returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE client
    SET
        shop_name = p_shop_name,
        email = p_email,
```

```

    phone_number = p_phone_number,
    address = p_shop_address
  WHERE
    shop_id = p_shop_id;

  RETURN FOUND;
END;
$$;

alter function fn_update_shop_profile_info(integer, text, text, text, text) owner
to root;

```

- **Result Structure:** Returns BOOLEAN (True if update successful).

```

1 ✓  SELECT fn_update_shop_profile_info(
2           p_shop_id 20, p_shop_name 'Sweet Cake Shop', p_email 'contact@sweetcake.com', p_phone_number '555-1234', p_shop_address '123 Baker Street'
3       );|
```

	fn_update_shop_profile_info
1	true

## fn\_list\_shop\_managers

- **Purpose:** Lists all platform managers and staff members who have login access to the specified shop.
- **How It Works (Logic):** Joins manager\_user with shop\_manager\_access using user\_id. The query is strictly filtered by shop\_id to ensure tenant isolation and only shows authorized personnel. It concatenates first and last names into a full\_name.
- **Code:**

```

create function fn_list_shop_managers(p_shop_id integer)
  returns TABLE(user_id integer, username character varying, full_name text)
  language plpgsql
as
$$
BEGIN
  RETURN QUERY
  SELECT
    u.user_id,
    u.username,
    (u.first_name || ' ' || u.last_name) AS full_name
  FROM "user" u
    JOIN shop_manager_access sma ON u.user_id = sma.manager_user_id
  WHERE sma.shop_id = p_shop_id -- CRITICAL: Tenant Isolation

```

```

        ORDER BY u.last_name, u.first_name;
    END;
$$;

alter function fn_list_shop_managers(integer) owner to root;

```

- Result Structure:** Returns the list of managers: (user\_id INTEGER, username VARCHAR, full\_name TEXT).
- 

Staff Members		5 team members		Add Staff
Sarah Johnson	active	sarah@sweetdreams.com	(555) 234-5678	Hired: 1/15/2023
Mike Chen	active	mike@sweetdreams.com	(555) 345-6789	Hired: 3/20/2023
Emily Rodriguez	active	emily@sweetdreams.com	(555) 456-7890	Hired: 6/10/2023
James Wilson	active	james@sweetdreams.com	(555) 567-8901	Hired: 8/5/2023
Lisa Taylor	active	lisa@sweetdreams.com	(555) 678-9012	Hired: 9/12/2023

## fn\_get\_shop\_staff\_members

- Purpose:** Retrieves a list of local (non-platform login) employees of the shop.
- How It Works (Logic):** Selects all fields from shop\_staff, filtering by shop\_id, and orders them alphabetically by name.
- Code:**

```

create function fn_get_shop_staff_members(p_shop_id integer)
    returns TABLE(staff_id integer, first_name text, last_name text, email text,
    phone_number text, job_title text, status text, hired_at date)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
    SELECT
        s.staff_id,
        s.first_name,
        s.last_name,
        s.email,
        s.phone_number,
        s.job_title,

```

```

        s.status,
        s.hired_at
    FROM
        shop_staff s
    WHERE
        s.shop_id = p_shop_id
    ORDER BY
        s.first_name, s.last_name;
END;
$$;

alter function fn_get_shop_staff_members(integer) owner to root;

```

- **Result Structure:** Returns local employee details: (staff\_id INTEGER, first\_name TEXT, last\_name TEXT, email TEXT, phone\_number TEXT, job\_title TEXT, status TEXT, hired\_at DATE).

SELECT * FROM fn_get_shop_staff_members( 20 );						
	staff_id	first_name	last_name	email	phone_number	job_title
1	12	Emily	Stone	emily.staff@example.com	555-4001	Cashier

The screenshot shows a web-based application interface for managing staff members. At the top, there is a navigation bar with a search icon and a user profile icon. Below the navigation, there is a section titled "Staff Members" which displays a list of five team members with their details: Sarah Johnson (active, Head Baker), Mike Chen (active, Pastry Chef), Emily Rodriguez (active, Cashier), James Wilson (active, Delivery Driver), and Lisa Taylor (active, Assistant Baker). Each staff member entry includes an email link, a phone number link, and a hire date. To the right of the staff list is a button labeled "Add Staff".

Below the staff list, there is a modal window titled "Add New Staff Member". This window contains fields for "Full Name \*", "Role/Position \*", "Email Address \*", "Phone Number", "Hire Date" (set to 30/11/2025), and a dropdown menu for "Active" status. There are also "Add Staff Member" and "Cancel" buttons at the bottom of the modal.

## fn\_add\_shop\_staff\_member

- **Purpose:** Adds a new local employee record to the shop\_staff table (for internal tracking/scheduling).
- **How It Works (Logic):** Inserts a new row into shop\_staff, setting default values for status ('active') and hired\_at (current date if not provided).
- **Code:**

```

create function fn_add_shop_staff_member(p_shop_id integer, p_first_name text,
                                         p_last_name text, p_email text, p_phone_number text, p_job_title text, p_hired_at
                                         date DEFAULT CURRENT_DATE) returns integer
    language plpgsql
as
$$
DECLARE
    v_new_staff_id INT;
BEGIN
    INSERT INTO shop_staff (
        shop_id,
        first_name,
        last_name,
        email,
        phone_number,
        job_title,
        status,
        hired_at
    )
    VALUES (
        p_shop_id,
        p_first_name,
        p_last_name,
        p_email,
        p_phone_number,
        p_job_title,
        'active',
        p_hired_at
    )
    RETURNING staff_id INTO v_new_staff_id;

    RETURN v_new_staff_id;
END;
$$;

alter function fn_add_shop_staff_member(integer, text, text, text, text, text,
                                         date) owner to root;

```

- **Result Structure:** Returns the ID of the new local staff record: INTEGER.

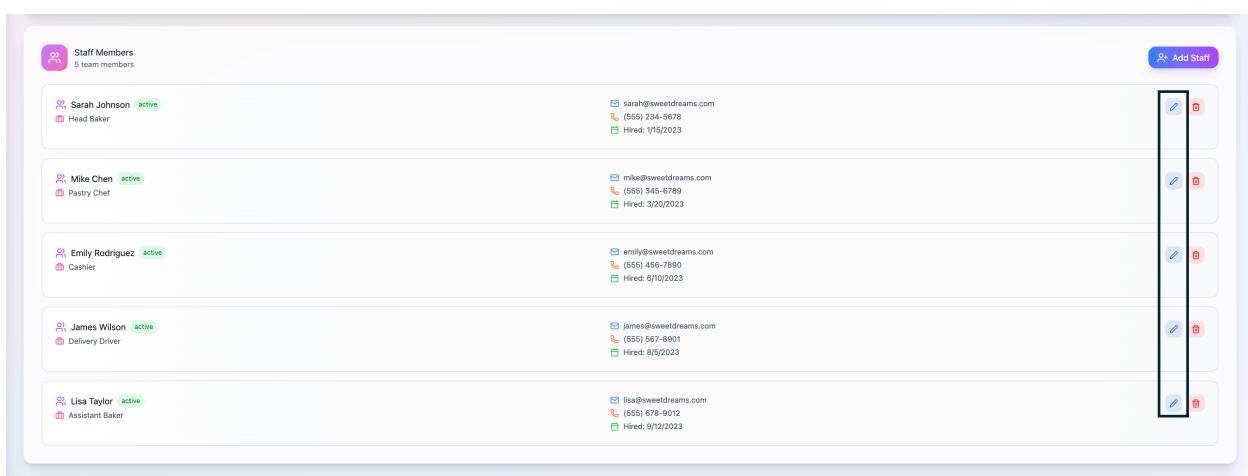


The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓ SELECT fn_add_shop_staff_member(
2           p_shop_id 20, p_first_name 'Sarah', p_last_name 'Lopez', p_email 'sarah@shop.com', p_phone_number '0909111222', p_job_title 'Decorator'
3       );
```

The output window shows the result of the function call:

	fn_add_shop_staff_member	Output
1	14	:integer

The screenshot shows a web-based staff management system. The title is "Staff Members" and it lists 5 team members:

- Sarah Johnson (active) - Head Baker
- Mike Chen (active) - Pastry Chef
- Emily Rodriguez (active) - Cashier
- James Wilson (active) - Delivery Driver
- Lisa Taylor (active) - Assistant Baker

Each staff member has a detailed view button and an edit icon (pencil) next to their name.


A modal dialog is open for Sarah Johnson, Head Baker. It displays her contact information:

- Name: Sarah Johnson
- Email: sarah@sweetdreams.com
- Phone: (555) 234-5678
- Hired: 1/15/2023

At the bottom left is a "Done" button.

## fn\_update\_shop\_staff\_member

- Purpose:** Modifies the details of an existing local employee record.
- How It Works (Logic):** Updates shop\_staff, ensuring both staff\_id and shop\_id match to prevent cross-shop modifications.
- Code:**

```
create function fn_update_shop_staff_member(p_staff_id integer, p_shop_id integer,
p_first_name text, p_last_name text, p_email text, p_phone_number text,
p_job_title text, p_status text, p_hired_at date) returns boolean
language plpgsql
as
$$
DECLARE
BEGIN
    UPDATE shop_staff
    SET
        first_name = p_first_name,
```

```

last_name = p_last_name,
email = p_email,
phone_number = p_phone_number,
job_title = p_job_title,
status = p_status,
hired_at = p_hired_at
WHERE
    staff_id = p_staff_id
    AND shop_id = p_shop_id;

IF FOUND THEN
    RETURN 1;
ELSE
    RETURN 0;
END IF;
END;
$$;

alter function fn_update_shop_staff_member(integer, integer, text, text, text,
text, text, date) owner to root;

```

- **Result Structure:** Returns BOOLEAN (True if update successful).



The screenshot shows a PostgreSQL terminal window. The command entered is:

```

1 ✓ SELECT fn_update_shop_staff_member(
2     p_staff_id 14, p_shop_id 20, p_first_name 'Sarah', p_last_name 'Lopez', p_email 'sarah@shop.com', p_phone_number '0909111222', p_job_title
3 );
4

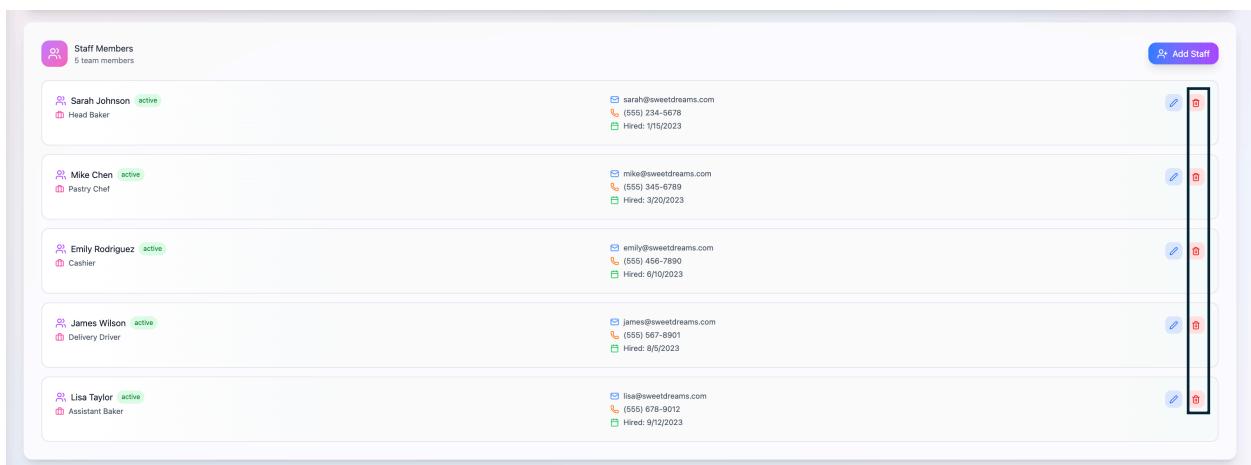
```

The output pane shows the result of the function call:

```

fn_update_shop_staff_member :boolean
1 true

```



The screenshot shows a web application interface titled "Staff Members". It displays a list of five team members:

- Sarah Johnson** (active) - Head Baker: Email: sarah@sweetdreams.com, Phone: (555) 234-5678, Hired: 1/15/2023
- Mike Chen** (active) - Pastry Chef: Email: mike@sweetdreams.com, Phone: (555) 345-6789, Hired: 3/20/2023
- Emily Rodriguez** (active) - Cashier: Email: emily@sweetdreams.com, Phone: (555) 456-7890, Hired: 6/10/2023
- James Wilson** (active) - Delivery Driver: Email: james@sweetdreams.com, Phone: (555) 567-8901, Hired: 8/8/2023
- Lisa Taylor** (active) - Assistant Baker: Email: lisa@sweetdreams.com, Phone: (555) 678-9012, Hired: 9/12/2023

Each staff member entry includes edit and delete icons.

## fn\_delete\_shop\_staff\_member

- **Purpose:** Deletes a local employee record, ensuring security checks are passed.
- **How It Works (Logic):** Deletes the row from shop\_staff where both staff\_id and shop\_id match.
- **Code:**

```

create function fn_delete_shop_staff_member(p_staff_id integer, p_shop_id integer)
returns boolean
language plpgsql
as
$$
DECLARE
    v_deleted_rows INT;
BEGIN
    DELETE FROM shop_staff
    WHERE
        staff_id = p_staff_id
        AND shop_id = p_shop_id; -- Security check

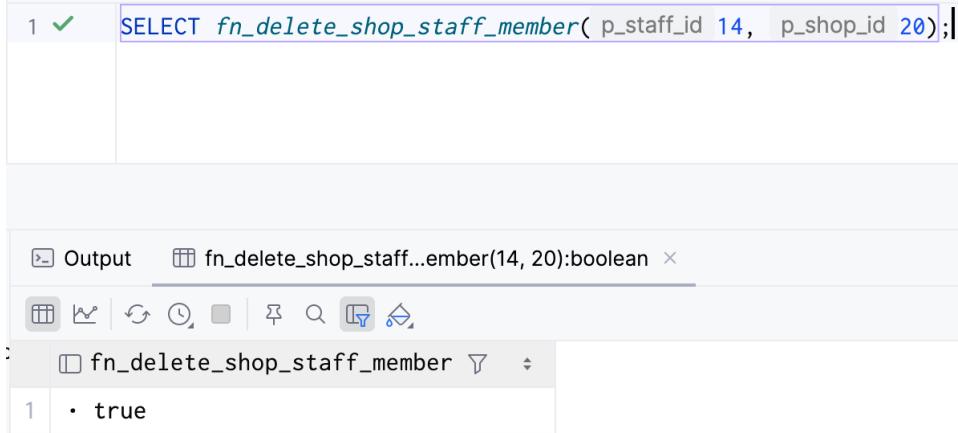
    GET DIAGNOSTICS v_deleted_rows = ROW_COUNT;

    RETURN FOUND;
END;
$$;

alter function fn_delete_shop_staff_member(integer, integer) owner to root;

```

- **Result Structure:** Returns BOOLEAN (True if delete successful).

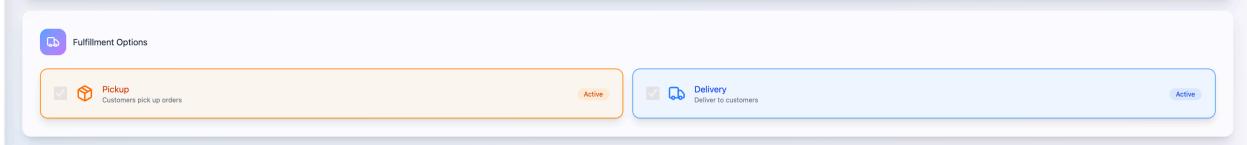


The screenshot shows a PostgreSQL terminal window. In the command line, a query is run:

```
1 ✓  SELECT fn_delete_shop_staff_member( p_staff_id 14,  p_shop_id 20);|
```

The output pane shows the results of the query:

```
Output fn_delete_shop_staff_member(14, 20):boolean ×
fn_delete_shop_staff_member
1 • true
```



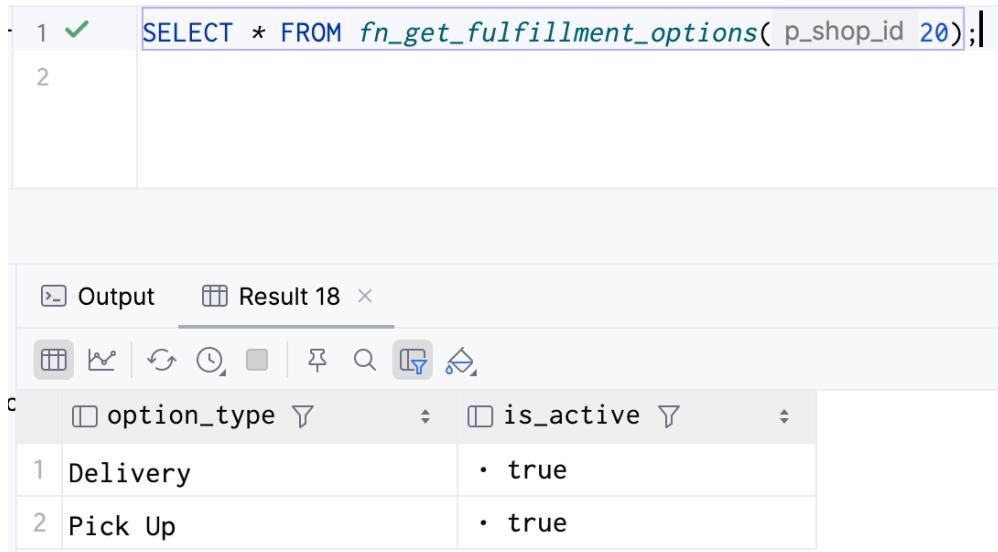
## fn\_get\_fulfillment\_options

- Purpose:** Lists the available order fulfillment options (e.g., Pick Up, Delivery) for a shop.
- How It Works (Logic):** Selects type and is\_active from fulfillment\_option filtered by shop\_id and ordered by type.
- Code:**

```
create function fn_get_fulfillment_options(p_shop_id integer)
    returns TABLE(option_type text, is_active boolean)
    language plpgsql
as
$$
DECLARE
BEGIN
    RETURN QUERY
        SELECT
            fo.type AS option_type,
            fo.is_active
        FROM
            fulfillment_option fo
        WHERE
            fo.shop_id = p_shop_id
        ORDER BY
            fo.type;
END;
$$;

alter function fn_get_fulfillment_options(integer) owner to root;
```

- Result Structure:** Returns the list of available methods: (option\_type TEXT, is\_active BOOLEAN).

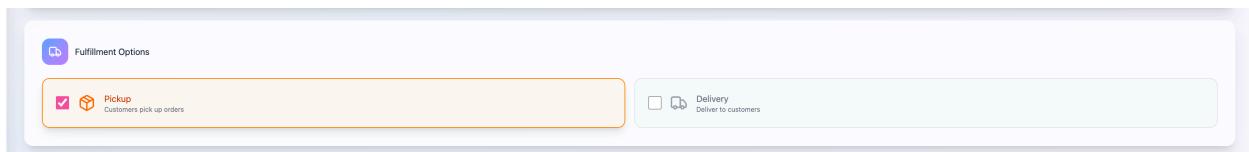


```

1 ✓
2
SELECT * FROM fn_get_fulfillment_options( p_shop_id 20);|
```

Output Result 18

	option_type	is_active
1	Delivery	• true
2	Pick Up	• true



## fn\_update\_fulfillment\_option\_status

- **Purpose:** Activates or deactivates a fulfillment option, or creates it if it does not yet exist (UPSERT logic).
- **How It Works (Logic):** Attempts an UPDATE on fulfillment\_option. If NOT FOUND (meaning the option doesn't exist yet for that shop), it performs an INSERT. This guarantees the option is always set to the desired status.
- **Code:**
- ```

create function fn_update_fulfillment_option_status(p_shop_id integer, p_type
text, p_is_active boolean) returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE fulfillment_option
    SET is_active = p_is_active
    WHERE shop_id = p_shop_id AND type = p_type;

    IF NOT FOUND THEN
        INSERT INTO fulfillment_option (shop_id, type, is_active)
        VALUES (p_shop_id, p_type, p_is_active);
    END IF;
END;
$$

```

```

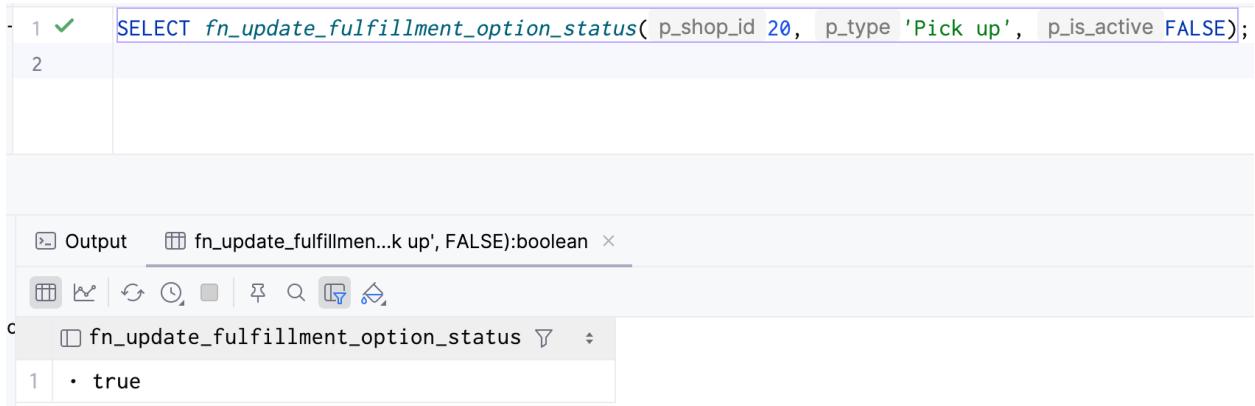
    END IF;

    RETURN TRUE;
END;
$$;

alter function fn_update_fulfillment_option_status(integer, text, boolean) owner
to root;

```

- **Result Structure:** Returns BOOLEAN (Always TRUE if the transaction completes).



The screenshot shows a PostgreSQL query editor interface. A query is run:

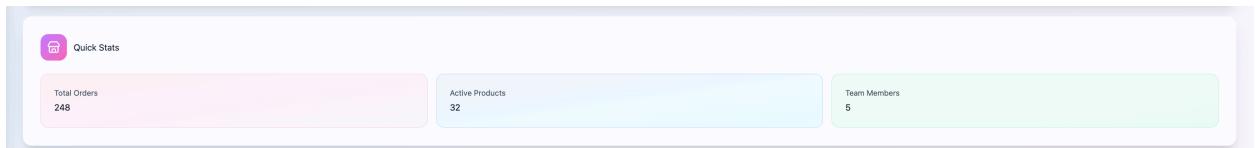
```

1 ✓ SELECT fn_update_fulfillment_option_status( p_shop_id 20, p_type 'Pick up', p_is_active FALSE);
2

```

The result is displayed in the 'Output' tab:

|   | fn_update_fulfillment_option_status(p_shop_id 20, p_type 'Pick up', p_is_active FALSE):boolean |
|---|------------------------------------------------------------------------------------------------|
| 1 | • true                                                                                         |



### **fn\_get\_shop\_quick\_stats**

- **Purpose:** Fetches key summary metrics for the shop dashboard (e.g., for quick access cards).
- **How It Works (Logic):** Executes three separate COUNT queries: total orders, total active products, and total active team members (shop\_staff with status='active').
- **Code:**

```

create function fn_get_shop_quick_stats(p_shop_id integer)
    returns TABLE(total_orders bigint, active_products bigint, team_members
    bigint)
    language plpgsql
as
$$
DECLARE
    v_total_orders BIGINT;

```

```

v_active_products BIGINT;
v_team_members BIGINT;
BEGIN
    SELECT COUNT(*) INTO v_total_orders
    FROM "order"
    WHERE shop_id = p_shop_id;

    SELECT COUNT(*) INTO v_active_products
    FROM product
    WHERE shop_id = p_shop_id
        AND is_active = TRUE;

    SELECT COUNT(*) INTO v_team_members
    FROM shop_staff
    WHERE shop_id = p_shop_id
        AND status = 'active';

    RETURN QUERY
        SELECT
            v_total_orders,
            v_active_products,
            v_team_members;
END;
$$;

```

```
alter function fn_get_shop_quick_stats(integer) owner to root
```

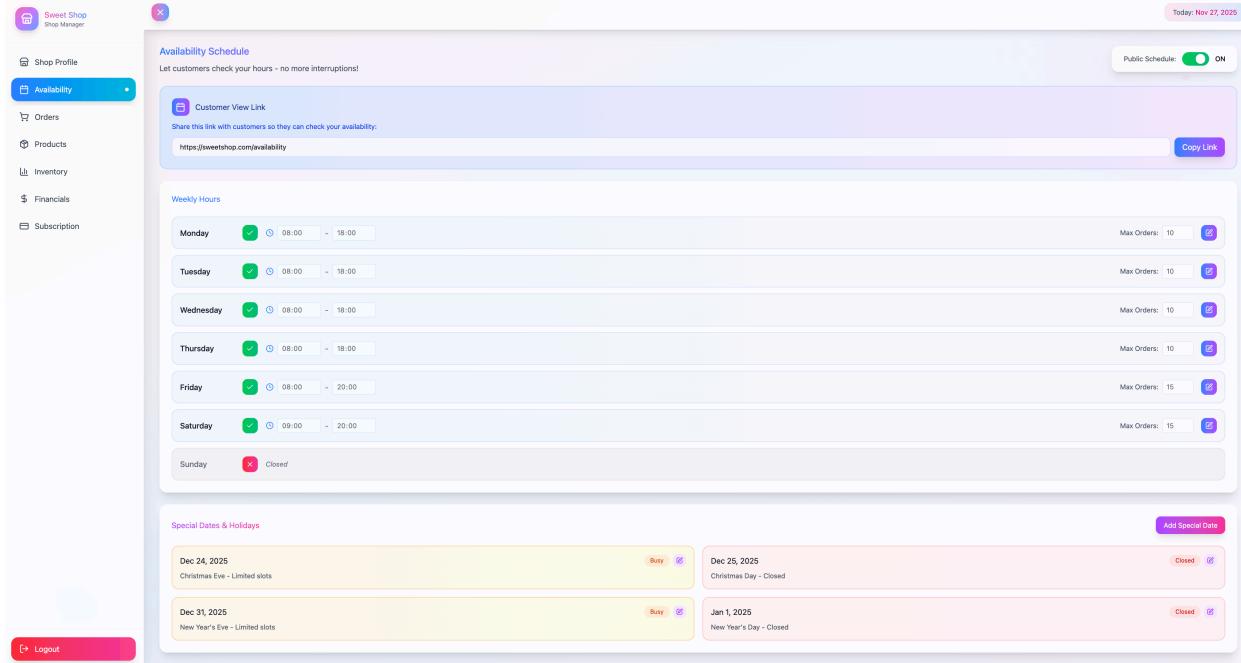
- **Result Structure:** Returns three aggregate counts: (total\_orders BIGINT, active\_products BIGINT, team\_members BIGINT).



The screenshot shows a PostgreSQL query editor interface. At the top, there is a code editor window containing the SQL code for the function definition. Below it is a results window titled 'Result 20' which displays the output of the function execution.

|   | total_orders | active_products | team_members |
|---|--------------|-----------------|--------------|
| 1 | 1            | 1               | 1            |

## Scheduling and Availability



## fn\_get\_shop\_availability

- Purpose:** Retrieves the standard weekly opening hours and order limits for a shop.
- How It Works (Logic):** Selects availability data and uses a CASE statement in the ORDER BY clause to sort the days chronologically (Monday=1, Sunday=7).
- Code:**

```
create function fn_get_shop_availability(p_shop_id integer)
    returns TABLE(day_of_week text, is_open boolean, open_time time without time
zone, close_time time without time zone, max_orders integer)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            sa.day_of_week,
            sa.is_open,
            sa.open_time,
            sa.close_time,
            sa.max_orders
        FROM
            shop_availability sa
        WHERE
            sa.shop_id = p_shop_id
        ORDER BY
            CASE
                WHEN sa.day_of_week = 'Monday' THEN 1
                WHEN sa.day_of_week = 'Tuesday' THEN 2
                WHEN sa.day_of_week = 'Wednesday' THEN 3
                WHEN sa.day_of_week = 'Thursday' THEN 4
                WHEN sa.day_of_week = 'Friday' THEN 5
                WHEN sa.day_of_week = 'Saturday' THEN 6
                WHEN sa.day_of_week = 'Sunday' THEN 7
            END
            , sa.open_time;
END
$$
```

```

WHEN sa.day_of_week = 'Tuesday' THEN 2
WHEN sa.day_of_week = 'Wednesday' THEN 3
WHEN sa.day_of_week = 'Thursday' THEN 4
WHEN sa.day_of_week = 'Friday' THEN 5
WHEN sa.day_of_week = 'Saturday' THEN 6
WHEN sa.day_of_week = 'Sunday' THEN 7
END;
$$;

```

```
alter function fn_get_shop_availability(integer) owner to root;
```

- **Result Structure:** Returns the weekly schedule: (day\_of\_week TEXT, is\_open BOOLEAN, open\_time TIME, close\_time TIME, max\_orders INTEGER).

1 ✓ `SELECT * FROM fn_get_shop_availability( p_shop_id 20);`

2

Output Result 21 ×

day\_of\_week is\_open open\_time close\_time max\_orders

| day_of_week | is_open | open_time | close_time | max_orders |
|-------------|---------|-----------|------------|------------|
| Monday      | true    | 09:00:00  | 18:00:00   | 20         |
| Tuesday     | true    | 09:00:00  | 18:00:00   | 20         |

The screenshot shows the Sweet Shop Shop Manager interface. On the left, there's a sidebar with options like Shop Profile, Availability (which is selected and highlighted in blue), Orders, Products, Inventory, Financials, Subscription, and Logout. The main content area is titled "Availability Schedule" and says "Let customers check your hours - no more interruptions!" It has a "Customer View Link" section with a link "https://sweetshop.com/availability" and a "Copy Link" button. Below that is the "Weekly Hours" section, which lists days from Monday to Saturday with their respective opening and closing times and maximum order limits. A "Public Schedule" toggle switch is turned on. At the bottom, there's a "Special Dates & Holidays" section with entries for Dec 24, 2025 (Christmas Eve - Limited slots), Dec 25, 2025 (Christmas Day - Closed), Dec 31, 2025 (New Year's Eve - Limited slots), and Jan 1, 2026 (New Year's Day - Closed). The "Add Special Date" button is visible.

## fn\_update\_day\_availability

- **Purpose:** Modifies or creates the standard schedule entry for a specific day (UPSERT logic).
- **How It Works (Logic):** Executes an UPDATE based on shop\_id and day\_of\_week. If NOT FOUND, it performs an INSERT, ensuring a rule exists for that day.
- **Code:**

```

create function fn_update_day_availability(p_shop_id integer, p_day_of_week text,
p_is_open boolean, p_open_time time without time zone, p_close_time time without
time zone, p_max_orders integer) returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE shop_availability
    SET
        is_open = p_is_open,
        open_time = p_open_time,
        close_time = p_close_time,
        max_orders = p_max_orders,
        updated_at = NOW()
    WHERE
        shop_id = p_shop_id
        AND day_of_week = p_day_of_week;

    IF NOT FOUND THEN
        INSERT INTO shop_availability (
            shop_id, day_of_week, is_open, open_time, close_time, max_orders,
updated_at
        )
        VALUES (
            p_shop_id, p_day_of_week, p_is_open, p_open_time, p_close_time,
p_max_orders, NOW()
        );
    END IF;

    RETURN TRUE;
END;
$$;

alter function fn_update_day_availability(integer, text, boolean, time, time,
integer) owner to root;

```

- **Result Structure:** Returns BOOLEAN.

```

1 ✓ SELECT fn_update_day_availability(
2           p_shop_id 20, p_day_of_week 'Monday', p_is_open TRUE, p_open_time '09:00', p_close_time '17:00', p_max_orders 20
3       );
4

```

Output fn\_update\_day\_availability(20) :boolean

fn\_update\_day\_availability

1 · true

| Special Dates & Holidays                       |        |
|------------------------------------------------|--------|
| Dec 24, 2025<br>Christmas Eve - Limited slots  | Busy   |
| Dec 25, 2025<br>Christmas Day - Closed         | Closed |
| Dec 31, 2025<br>New Year's Eve - Limited slots | Busy   |
| Jan 1, 2026<br>New Year's Day - Closed         | Closed |

## fn\_get\_shop\_special\_dates

- Purpose:** Lists scheduled non-standard dates (holidays, special hours) for shop management.
- How It Works (Logic):** Selects all special date data filtered by shop\_id and orders them chronologically.
- Code:**

```

create function fn_get_shop_special_dates(p_shop_id integer)
    returns TABLE(special_date_id integer, date date, status text, note text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            sd.special_date_id,
            sd.date,
            sd.status,
            sd.note
        FROM
            shop_special_date sd
        WHERE
            sd.shop_id = p_shop_id
        ORDER BY
            sd.date ASC;
END;
$$;

```

```
alter function fn_get_shop_special_dates(integer) owner to root;
```

- **Result Structure:** Returns the special date list: (special\_date\_id INTEGER, date DATE, status TEXT, note TEXT).

1 ✓ `SELECT * FROM fn_get_shop_special_dates( p_shop_id 20);`

2

|   | special_date_id | date       | status | note    |
|---|-----------------|------------|--------|---------|
| 1 | 9               | 2025-12-04 | Closed | Holiday |

The top screenshot shows a list of special dates with columns for Date, Status, and Note. The bottom screenshot shows a detailed view of a specific date entry with fields for Date, Status, and Note, along with a Save button.

## fn\_upsert\_shop\_special\_date

- **Purpose:** Modifies or creates a special date entry (UPSERT logic).
- **How It Works (Logic):** Executes an UPDATE based on shop\_id and date. If NOT FOUND, performs an INSERT.
- **Code:**

```

create function fn_upsert_shop_special_date(p_shop_id integer, p_date date,
p_status text, p_note text) returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE shop_special_date
    SET
        status = p_status,
        note = p_note,
        updated_at = NOW()
    WHERE
        shop_id = p_shop_id
        AND date = p_date;

    IF NOT FOUND THEN
        INSERT INTO shop_special_date (shop_id, date, status, note)
        VALUES (p_shop_id, p_date, p_status, p_note);
    END IF;

    RETURN TRUE;
END;
$$;

alter function fn_upsert_shop_special_date(integer, date, text, text) owner to
root;

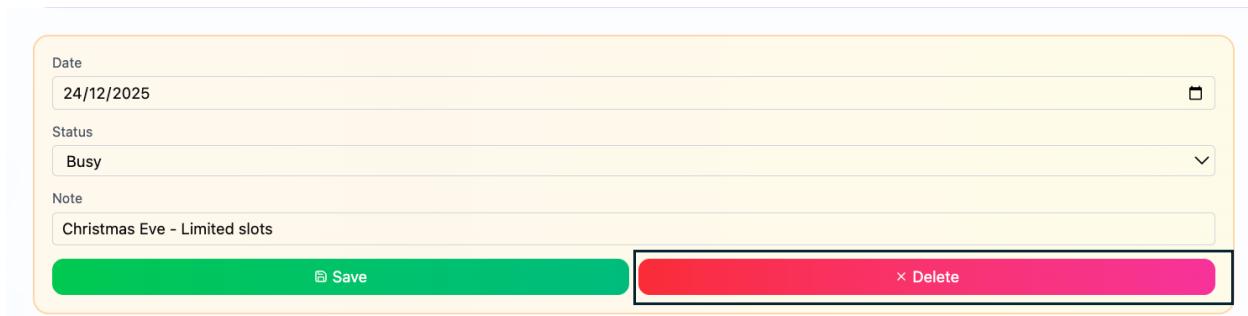
```

- **Result Structure:** Returns BOOLEAN.

1 ✓ | `SELECT fn_upsert_shop_special_date( p_shop_id 20, p_date '2024-12-25', p_status 'Closed', p_note 'Christmas Day');`

2 |

A screenshot of a PostgreSQL terminal window. The command `SELECT fn\_upsert\_shop\_special\_date( p\_shop\_id 20, p\_date '2024-12-25', p\_status 'Closed', p\_note 'Christmas Day');` is entered at the prompt. The output shows the result of the function call: `fn\_upsert\_shop\_special\_date(boolean)` with a value of `true`.



A screenshot of a web-based form for creating a new shop special date entry. The form fields are:

- Date: 24/12/2025
- Status: Busy
- Note: Christmas Eve - Limited slots
- Buttons: Save (green) and Delete (red)

## fn\_delete\_shop\_special\_date

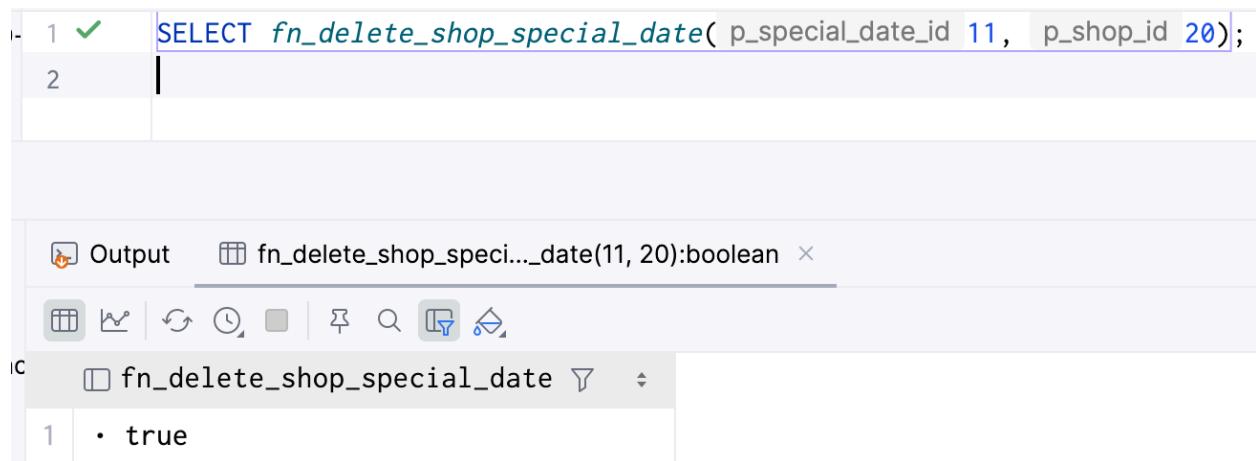
- **Purpose:** Removes a special date entry.
- **How It Works (Logic):** Executes a standard DELETE query, requiring both special\_date\_id and shop\_id for security.
- **Code:**

```
create function fn_delete_shop_special_date(p_special_date_id integer, p_shop_id
integer) returns boolean
language plpgsql
as
$$
BEGIN
    DELETE FROM shop_special_date
    WHERE
        special_date_id = p_special_date_id
        AND shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

alter function fn_delete_shop_special_date(integer, integer) owner to root;
```

- **Result Structure:** Returns BOOLEAN.



The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓ SELECT fn_delete_shop_special_date( p_special_date_id 11, p_shop_id 20);
```

The output window shows the result of the function execution:

```
Output fn_delete_shop_speci..._date(11, 20):boolean
fn_delete_shop_special_date
1 · true
```

| Weekly Hours |                                     |        |         |
|--------------|-------------------------------------|--------|---------|
| Monday       | <input checked="" type="checkbox"/> | 08:00  | - 18:00 |
| Tuesday      | <input checked="" type="checkbox"/> | 08:00  | - 18:00 |
| Wednesday    | <input checked="" type="checkbox"/> | 08:00  | - 18:00 |
| Thursday     | <input checked="" type="checkbox"/> | 08:00  | - 18:00 |
| Friday       | <input checked="" type="checkbox"/> | 08:00  | - 20:00 |
| Saturday     | <input checked="" type="checkbox"/> | 09:00  | - 20:00 |
| Sunday       | <input checked="" type="checkbox"/> | Closed |         |

## fn\_get\_shop\_today\_hours

- Purpose:** Retrieves and formats the shop's hours for the current day for real-time display.
- How It Works (Logic):** Determines the current day's name, queries shop\_availability, and uses a CASE statement to format the times into a user-friendly string (e.g., "8:00 AM - 4:00 PM").
- Code:**

```

create function fn_get_shop_today_hours(p_shop_id integer)
    returns TABLE(day_name text, is_open boolean, time_display text, status_label
text)
        language plpgsql
as
$$
DECLARE
    v_today_name TEXT;
BEGIN
    SELECT TRIM(TO_CHAR(CURRENT_DATE, 'Day')) INTO v_today_name;

    RETURN QUERY
        SELECT
            sa.day_of_week::TEXT,
            sa.is_open,
            CASE
                WHEN sa.is_open THEN
                    TO_CHAR(sa.open_time, 'FMHH12:MI AM') || ' - ' ||
                    TO_CHAR(sa.close_time, 'FMHH12:MI PM')
                ELSE 'Closed'
            END::TEXT AS time_display,
            CASE
                WHEN sa.is_open = FALSE THEN 'Closed'
                WHEN sa.day_of_week IN ('Saturday', 'Sunday') THEN 'Weekend hours'
                ELSE 'Regular hours'
            END::TEXT AS status_label
        FROM
            shop_availability sa
        WHERE
            p_shop_id = sa.shop_id;
END;
$$
language plpgsql;

```

```

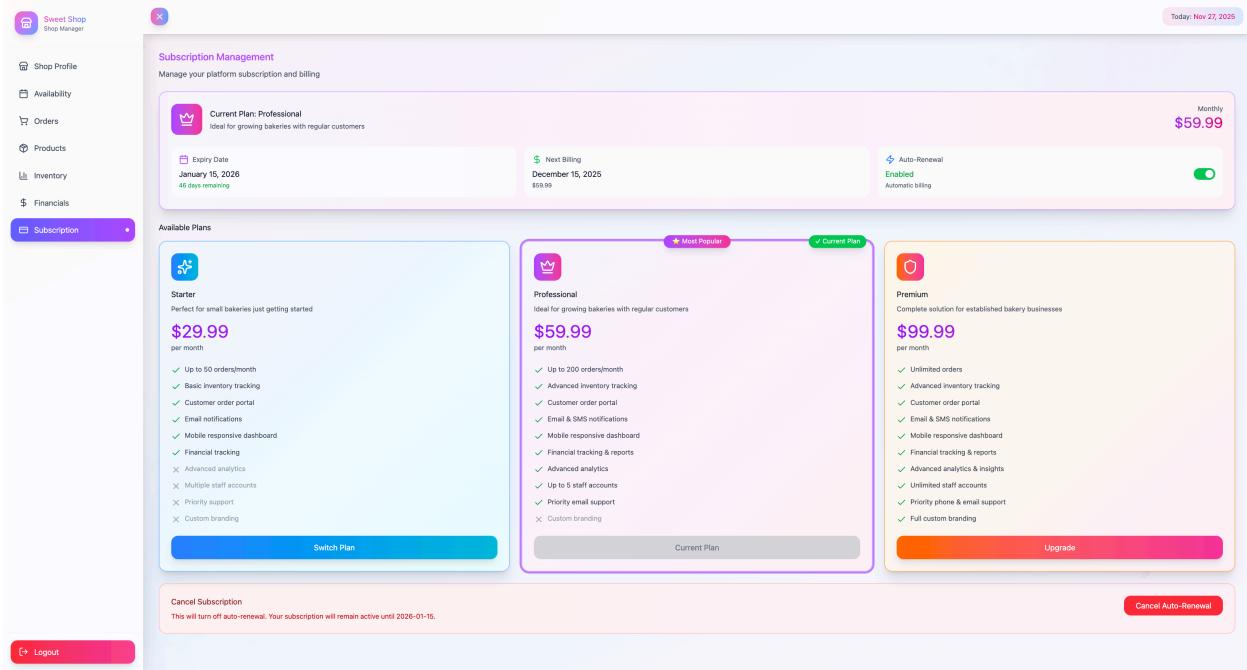
    sa.shop_id = p_shop_id
    AND sa.day_of_week = v_today_name;
END;
$$;

alter function fn_get_shop_today_hours(integer) owner to root;

```

- Result Structure:** Returns today's specific status: (day\_name TEXT, is\_open BOOLEAN, time\_display TEXT, status\_label TEXT).

## 4. Subscriptions (Platform Level)



### fn\_get\_shop\_subscription

- Purpose:** Retrieves the client's current subscription details, combining static plan data with the shop's dynamic billing status.
- How It Works (Logic):** Joins client (for snapshot price, cycle, and status) with platform\_plans (for the current plan name). Calculates days\_remaining until the next billing date.
- Code:**

```

create function fn_get_shop_subscription(p_shop_id integer)
    returns TABLE(plan_name text, plan_price numeric, billing_cycle text,
next_billing_date date, is_auto_renewal boolean, status text, expiry_date date,
days_remaining integer)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.name::TEXT AS plan_name,          -- Alias explicitly to match return
            table
            c.plan_price,
            c.billing_cycle::TEXT,
            c.next_billing_date,
            c.is_auto_renewal_enabled,
            c.subscription_status::TEXT,
            c.next_billing_date AS expiry_date,
            (c.next_billing_date - CURRENT_DATE)::INT AS days_remaining
        FROM
            client c
                LEFT JOIN
            platform_plans p ON c.plan_id = p.plan_id
        WHERE
            c.shop_id = p_shop_id;
END;
$$;

alter function fn_get_shop_subscription(integer) owner to root;

```

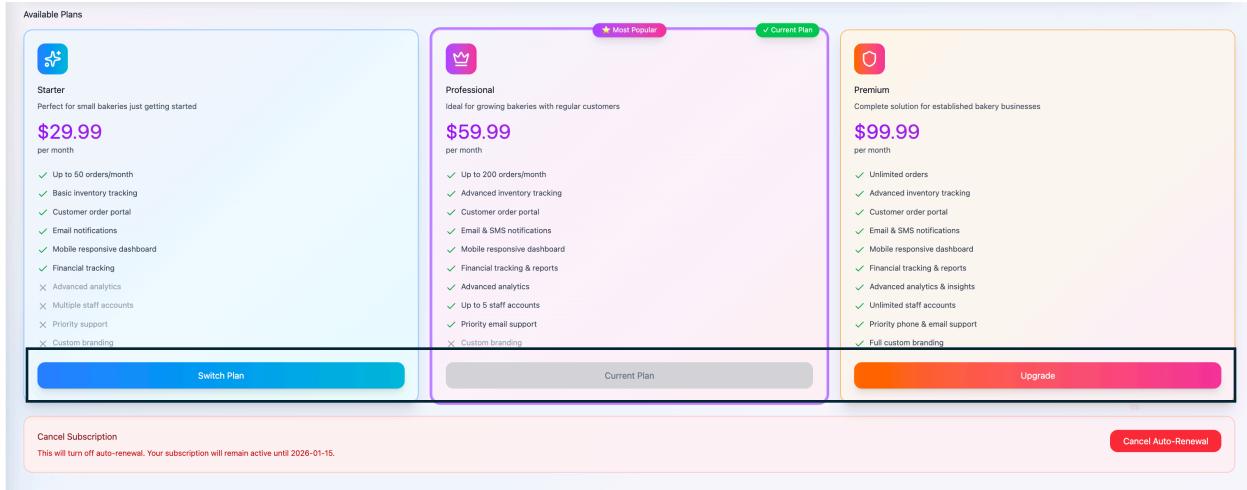
- **Result Structure:** Returns detailed subscription status: (plan\_name TEXT, plan\_price NUMERIC, billing\_cycle TEXT, next\_billing\_date DATE, is\_auto\_renewal BOOLEAN, status TEXT, expiry\_date DATE, days\_remaining INTEGER).

1 ✓ SELECT \* FROM fn\_get\_shop\_subscription( p\_shop\_id 20);

2

The screenshot shows a PostgreSQL terminal window. The command `SELECT \* FROM fn\_get\_shop\_subscription( p\_shop\_id 20);` is entered in the first line. The result set is displayed in the second line, showing one row of data. The data is presented in a table with columns: plan\_name, plan\_price, billing\_cycle, next\_billing\_date, is\_auto\_renewal, status, and expiry\_date. The row values are: Standard, 29.99, Monthly, 2025-12-29, true, Active, 2025-12-29.

| plan_name | plan_price | billing_cycle | next_billing_date | is_auto_renewal | status | expiry_date |
|-----------|------------|---------------|-------------------|-----------------|--------|-------------|
| Standard  | 29.99      | Monthly       | 2025-12-29        | true            | Active | 2025-12-29  |



## fn\_update\_subscription\_plan

- Purpose:** Changes a shop's subscription tier, capturing the current market price for billing integrity.
- How It Works (Logic):** Fetches the *current* `price_monthly` from `platform_plans`. It then updates `client`, setting the new `plan_id` and the captured `plan_price` (maintaining the snapshot billing model).
- Code:**

```

create function fn_update_subscription_plan(p_shop_id integer, p_new_plan_id
integer) returns boolean
language plpgsql
as
$$
DECLARE
v_new_price DECIMAL(10,2);
BEGIN
    -- Get current market price of the new plan
    SELECT price_monthly INTO v_new_price
    FROM platform_plans
    WHERE plan_id = p_new_plan_id;

    -- If Plan ID doesn't exist, return false
    IF NOT FOUND THEN
        RETURN FALSE;
    END IF;

    -- Update client with new plan ID and the NEW price
    UPDATE client
    SET
        plan_id = p_new_plan_id,
        plan_price = v_new_price
    WHERE

```

```

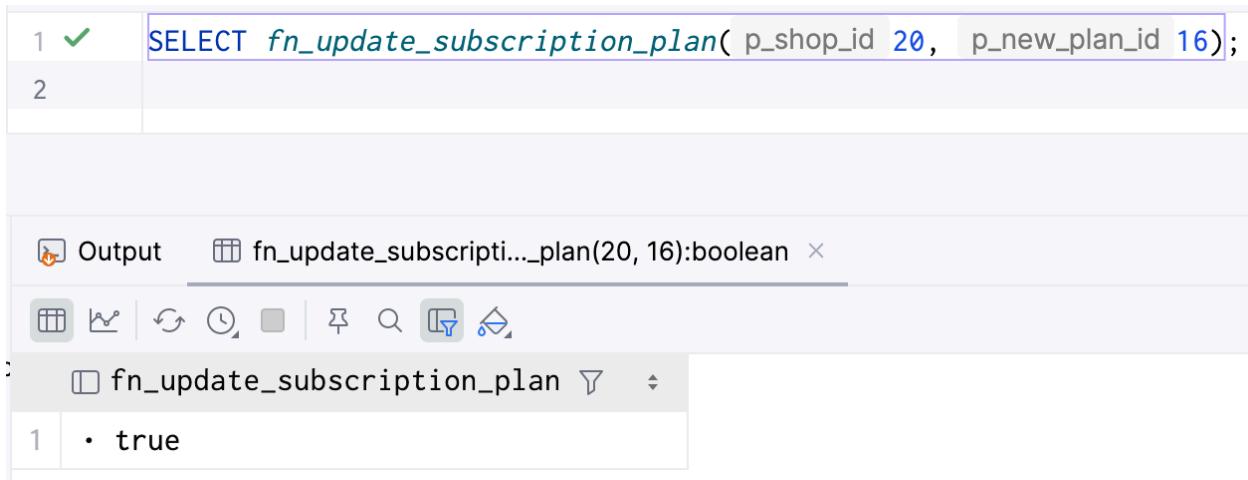
shop_id = p_shop_id;

RETURN FOUND;
END;
$$;

alter function fn_update_subscription_plan(integer, integer) owner to root;

```

- **Result Structure:** Returns BOOLEAN.

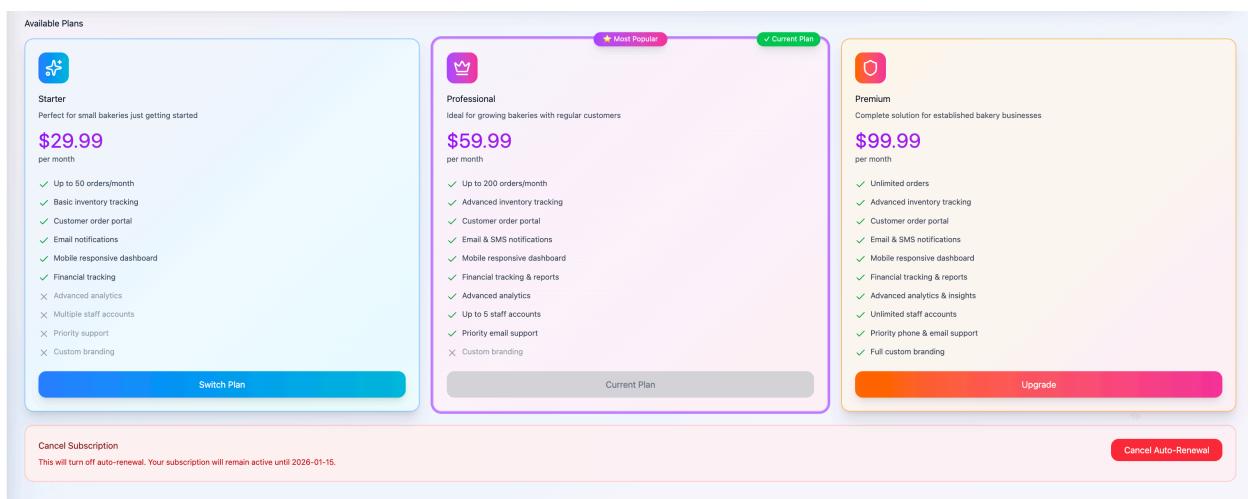


The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓ SELECT fn_update_subscription_plan( p_shop_id 20, p_new_plan_id 16);
```

The result of the query is displayed in the 'Output' tab:

```
fn_update_subscription_plan(20, 16):boolean
1 • true
```



## fn\_get\_available\_plans

- **Purpose:** Lists all available platform plans for upgrade/downgrade options, marking the shop's current tier.

- **How It Works (Logic):** Queries platform\_plans. It uses a comparison against the shop's current plan\_id to set the is\_current\_plan flag.

- **Code:**

```

create function fn_get_available_plans(p_shop_id integer)
    returns TABLE(plan_id integer, name text, price numeric, description text,
    features text[], is_popular boolean, is_current_plan boolean)
        language plpgsql
as
$$
DECLARE
    v_current_plan_id INTEGER;
BEGIN
    -- Get the shop's current plan ID
    SELECT c.plan_id INTO v_current_plan_id
    FROM client c
    WHERE c.shop_id = p_shop_id;

    RETURN QUERY
        SELECT
            p.plan_id,
            p.name::TEXT,
            p.price_monthly AS price,          -- Alias to match return table 'price'
            p.description::TEXT,
            p.features,
            p.is_popular,
            -- Safe boolean comparison
            COALESCE(p.plan_id = v_current_plan_id, FALSE) AS is_current_plan
        FROM
            platform_plans p
        ORDER BY
            p.price_monthly ASC;
END;
$$;

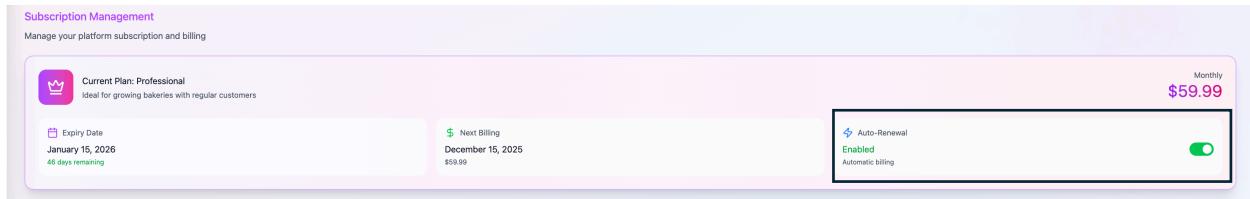
alter function fn_get_available_plans(integer) owner to root;

```

- **Result Structure:** Returns the plan catalog: (plan\_id INTEGER, name TEXT, price NUMERIC, description TEXT, features TEXT[], is\_popular BOOLEAN, is\_current\_plan BOOLEAN).

The screenshot shows a PostgreSQL terminal window. At the top, there is a command line input field containing the SQL query: `SELECT * FROM fn_get_available_plans( p_shop_id 20);`. Below the command line, the output pane displays the results of the query. The results are presented in a table with the following data:

|   | plan_id | name     | price | description   | features                           | is_popular | is_current_plan |
|---|---------|----------|-------|---------------|------------------------------------|------------|-----------------|
| 1 | 16      | Basic    | 19.99 | Starter plan  | {10 products, Basic support}       | true       | true            |
| 2 | 17      | Standard | 29.99 | Standard plan | {50 products, Priority support}    | true       | false           |
| 3 | 18      | Premium  | 49.99 | Premium plan  | {Unlimited products, 24/7 support} | false      | false           |



## fn\_toggle\_auto\_renewal

- **Purpose:** Enables or disables the subscription's auto-renewal setting.
- **How It Works (Logic):** Executes a simple UPDATE on the client table to set the `is_auto_renewal_enabled` flag.
- **Code:**

```
create function fn_toggle_auto_renewal(p_shop_id integer, p_is_enabled boolean)
returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE client
    SET is_auto_renewal_enabled = p_is_enabled
    WHERE shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

alter function fn_toggle_auto_renewal(integer, boolean) owner to root;
```

- **Result Structure:** Returns BOOLEAN.

```

1 ✓ SELECT fn_toggle_auto_renewal( p_shop_id 20, p_is_enabled TRUE);
2

```

Output fn\_toggle\_auto\_renewal(20, TRUE):boolean

fn\_toggle\_auto\_renewal

1 true

## Order Management

The screenshot displays the Sweet Shop Order Tracking & Management interface. On the left, a sidebar provides navigation links for Shop Profile, Availability, Orders (selected), Products, Inventory, Financials, and Subscription. The main content area features a header "Order Tracking & Management" with a sub-header "Never miss a deadline - track every order with automatic reminders". Below this, there are two prominent sections: "Urgent Deadlines!" (highlighted in orange) and "Pending Customer Confirmations" (highlighted in yellow). The "Urgent Deadlines" section lists ORD-001 for Sarah Johnson (Custom Birthday Cake, due 2025-12-01 at 10:00 AM) and ORD-002 for Mike Chen (Quote sent: \$450.00, Deposit: \$225.00). The "Pending Customer Confirmations" section lists ORD-001 for Sarah Johnson (Quote: \$125.00, Deposit (PAID): \$62.50) and ORD-002 for Mike Chen (Quote: \$450.00, Deposit (PAID): \$225.00). At the bottom, there are buttons for "List View" and "Calendar View", and a search bar with placeholder text "Search by order number or customer...".

## fn\_get\_shop\_orders

- Purpose:** Retrieves a comprehensive list of orders for the shop's dashboard, supporting filtering, searching, and visual urgency alerts.
- How It Works (Logic):** Joins order with customer\_user. Uses a subquery with string\_agg on order\_item to create a product\_summary. Includes complex filtering by status and search query (checking order ID and customer name). Orders results by urgency (is\_urgent flag) and deadline.

- **Code:**

```

create function fn_get_shop_orders(p_shop_id integer, p_status_filter text DEFAULT
'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(order_id integer, customer_name text, product_summary text,
total_amount numeric, deposit_amount numeric, deposit_status text, status text,
deadline timestamp with time zone, is_urgent boolean, created_at timestamp with
time zone)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            o.order_id,
            (cu.first_name || ' ' || cu.last_name)::TEXT AS customer_name,
            COALESCE(
                -- FIXED: Added 'oi' alias and used 'oi.order_id' to be explicit
                (SELECT string_agg(product_name, ', ') FROM order_item oi
WHERE oi.order_id = o.order_id),
                'No items'
            )::TEXT AS product_summary,
            o.total_amount,
            o.deposit_amount,
            o.deposit_status,
            o.status,
            o.deadline,
            (o.deadline IS NOT NULL AND o.deadline < NOW() + INTERVAL '7 days' AND
o.status NOT IN ('Finished', 'Cancelled')) AS is_urgent,
            o.created_at
        FROM
            "order" o
            LEFT JOIN
            customer_user cu ON o.customer_id = cu.customer_id
        WHERE
            o.shop_id = p_shop_id
            AND (
                p_status_filter IN ('All', 'All Orders')
                OR o.status = p_status_filter
            )
            AND (
                p_search_query = ''
                OR o.order_id::TEXT ILIKE '%' || p_search_query || '%'
                OR cu.first_name ILIKE '%' || p_search_query || '%'
                OR cu.last_name ILIKE '%' || p_search_query || '%'
            )
        ORDER BY
            (o.deadline IS NOT NULL AND o.deadline < NOW() + INTERVAL '7 days' AND
o.status NOT IN ('Finished', 'Cancelled')) DESC,
            o.deadline ASC,
            o.created_at DESC;

```

```

END;
$$;

alter function fn_get_shop_orders(integer, text, text) owner to root;

```

- Result Structure:** Returns the order list summary: (order\_id INTEGER, customer\_name TEXT, product\_summary TEXT, total\_amount NUMERIC, deposit\_amount NUMERIC, deposit\_status TEXT, status TEXT, deadline TIMESTAMP WITH TIME ZONE, is\_urgent BOOLEAN, created\_at TIMESTAMP WITH TIME ZONE).

The screenshot shows a PostgreSQL terminal window. The command entered is:

```

1 ✓ SELECT * FROM fn_get_shop_orders( p_shop_id 20, p_status_filter 'All', p_search_query '' );
2

```

The output shows a single row of data:

|   | order_id | customer_name | product_summary | total_amount | deposit_amount | deposit_status | status    | deadline                          |
|---|----------|---------------|-----------------|--------------|----------------|----------------|-----------|-----------------------------------|
| 1 | 40       | John Doe      | Handmade Candle | 45.99        | 10             | Paid           | Confirmed | 2025-12-01 21:45:39.442277 +00:00 |

The screenshot displays a web application interface for managing orders. At the top, there are navigation buttons for "List View" and "Calendar View". A dropdown menu shows "All Orders".

Three orders are listed:

- ORD-001**: Status: In Process, Urgent. Placed by Sarah Johnson on 2025-11-25. Total quote: \$125.00, deposit: \$62.50. Order items: 1x Custom Birthday Cake (Note: Chocolate cake with buttercream, pink and gold theme). Action button: "Mark as Finished".
- ORD-002**: Status: Awaiting Confirmation, Urgent. Placed by Mike Chen on 2025-11-27. Total quote: \$450.00, deposit: \$225.00. Order items: 1x Wedding Cake - 3 Tier (Note: Vanilla and red velvet tiers, white fondant). Action button: "Resend Confirmation Link".
- ORD-003**: Status: Awaiting Quote, Urgent. Placed by Emily Davis on 2025-11-27. Total quote: \$150.00, deposit: \$75.00. Order items: 6x Custom Cookie Boxes (Note: Corporate event, company logo on cookies). Action button: "Send Quote".

Each order card includes a "Details" button and a vertical timeline on the right showing the status history.

## fn\_get\_order\_details

- Purpose:** Fetches the primary order and customer contact details necessary for order fulfillment.
- How It Works (Logic):** Joins order with customer\_user (LEFT JOIN to support guest orders).

- **Code:**

```

create function fn_get_order_details(p_order_id integer)
    returns TABLE(order_id integer, status text, deadline timestamp with time
zone, created_at timestamp with time zone, total_amount numeric, deposit_amount
numeric, deposit_status text, customer_first_name text, customer_last_name text,
customer_email text, customer_phone text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            o.order_id,
            o.status,
            o.deadline,
            o.created_at,
            o.total_amount,
            o.deposit_amount,
            o.deposit_status,
            cu.first_name::TEXT,
            cu.last_name::TEXT,
            cu.email::TEXT,
            cu.phone_number::TEXT
        FROM
            "order" o
                LEFT JOIN
            customer_user cu ON o.customer_id = cu.customer_id
        WHERE
            o.order_id = p_order_id;
END;
$$;

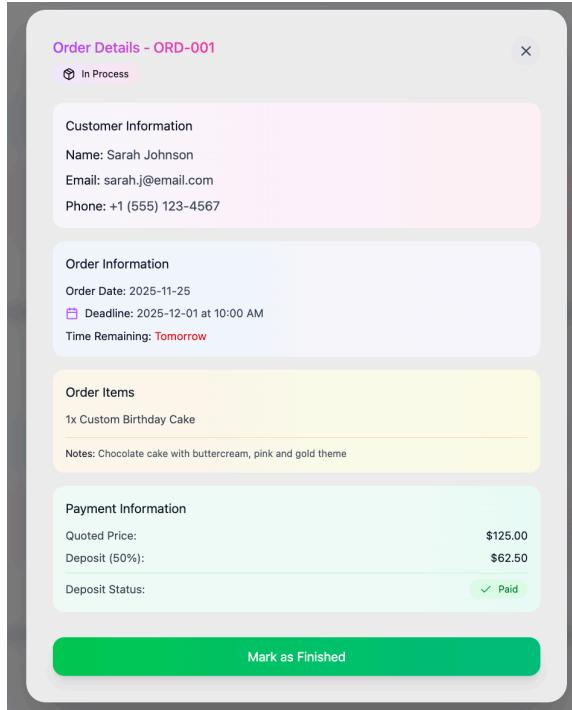
alter function fn_get_order_details(integer) owner to root;

```

- **Result Structure:** Returns detailed order status and customer contacts: (order\_id INTEGER, status TEXT, deadline TIMESTAMP WITH TIME ZONE, created\_at TIMESTAMP WITH TIME ZONE, total\_amount NUMERIC, deposit\_amount NUMERIC, deposit\_status TEXT, customer\_first\_name TEXT, customer\_last\_name TEXT, customer\_email TEXT, customer\_phone TEXT).

The screenshot shows a PostgreSQL database client interface. In the top-left, there is a code editor window containing the SQL query: `SELECT * FROM fn_get_order_details(41);`. Below it is a results pane titled "Result 37". The results table has the following data:

| order_id | status                | deadline                          | created_at                        | total_amount | deposit_amount | deposit_status | customer_first_name | customer_last_name |
|----------|-----------------------|-----------------------------------|-----------------------------------|--------------|----------------|----------------|---------------------|--------------------|
| 41       | Awaiting Confirmation | 2025-11-30 21:45:39.442277 +00:00 | 2025-11-29 21:45:39.442277 +00:00 | 25.5         | 0              | Pending        | Tom                 |                    |



## fn\_get\_order\_items

- Purpose:** Retrieves the list of line items (products, quantities, notes) associated with a specific order.]
- How It Works (Logic):** Simple query against order\_item filtered by order\_id.
- Code:**

```
create function fn_get_order_items(p_order_id integer)
    returns TABLE(product_name text, quantity integer, notes text, price numeric)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            oi.product_name,
            oi.quantity,
            oi.notes,
            oi.price
        FROM
            order_item oi
        WHERE
            oi.order_id = p_order_id;
END;
$$;
```

```
alter function fn_get_order_items(integer) owner to root;
```

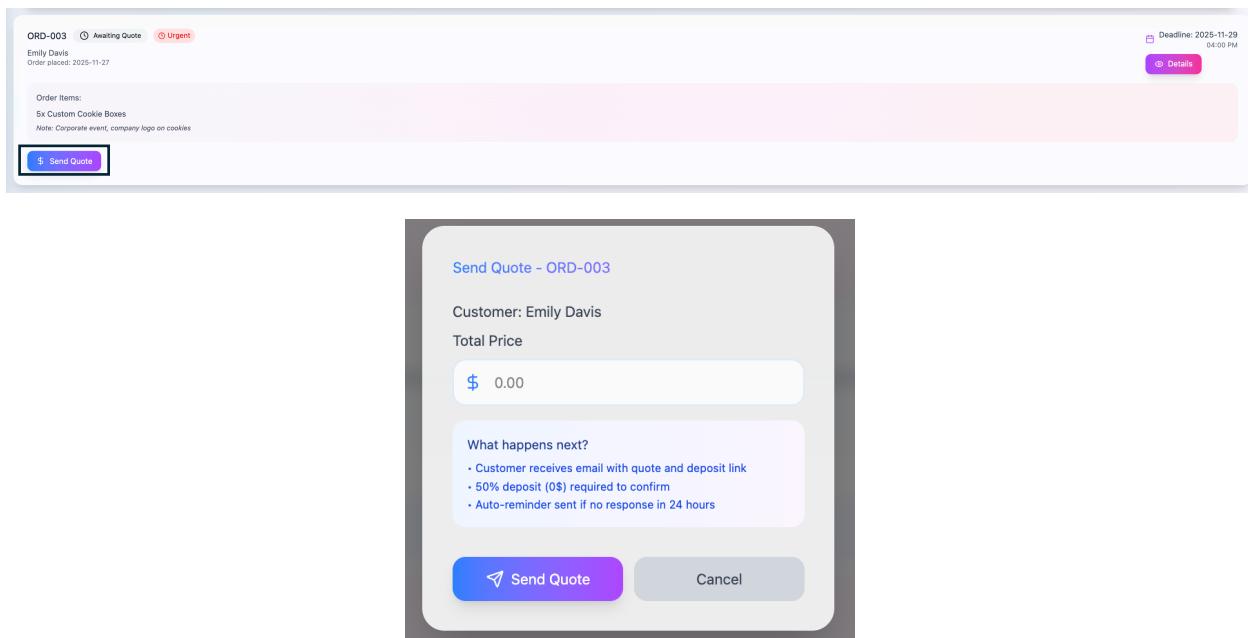
- **Result Structure:** Returns the order contents: (product\_name TEXT, quantity INTEGER, notes TEXT, price NUMERIC).

The screenshot shows a PostgreSQL database interface. A query is run:

```
1 ✓ SELECT * FROM fn_get_order_items( p_order_id 41);
```

The result table has four columns: product\_name, quantity, notes, and price. One row is returned:

| product_name   | quantity | notes              | price |
|----------------|----------|--------------------|-------|
| Wireless Mouse | 1        | Test before pickup | 25.5  |



## fn\_send\_order\_quote

- **Purpose:** Finalizes the price calculation for an order and prompts the customer to confirm and pay.
- **How It Works (Logic):** Updates order fields with the quoted total\_amount and required deposit\_amount, setting the status to 'Awaiting Confirmation' and deposit\_status to 'Pending'.

- **Code:**

```

create function fn_send_order_quote(p_order_id integer, p_total_amount numeric,
p_deposit_amount numeric) returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE "order"
    SET
        total_amount = p_total_amount,
        deposit_amount = p_deposit_amount,
        status = 'Awaiting Confirmation',
        deposit_status = 'Pending'
    WHERE
        order_id = p_order_id;

    RETURN FOUND;
END;
$$;

alter function fn_send_order_quote(integer, numeric, numeric) owner to root;

```

- **Result Structure:** Returns BOOLEAN.

```

1 ✓ SELECT fn_send_order_quote( p_order_id 41, p_total_amount 150.00, p_deposit_amount 50.00);
2

```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console

Tx Output fn\_send\_order\_quote(...50.00, 50.00):boolean ×

fn\_send\_order\_quote ↴

|   |        |
|---|--------|
| 1 | • true |
|---|--------|



## fn\_update\_order\_status

- **Purpose:** Updates an order's status based on workflow progression (e.g., 'Confirmed', 'In Process').

- **How It Works (Logic):** Executes a simple UPDATE on the order table to change the status field.

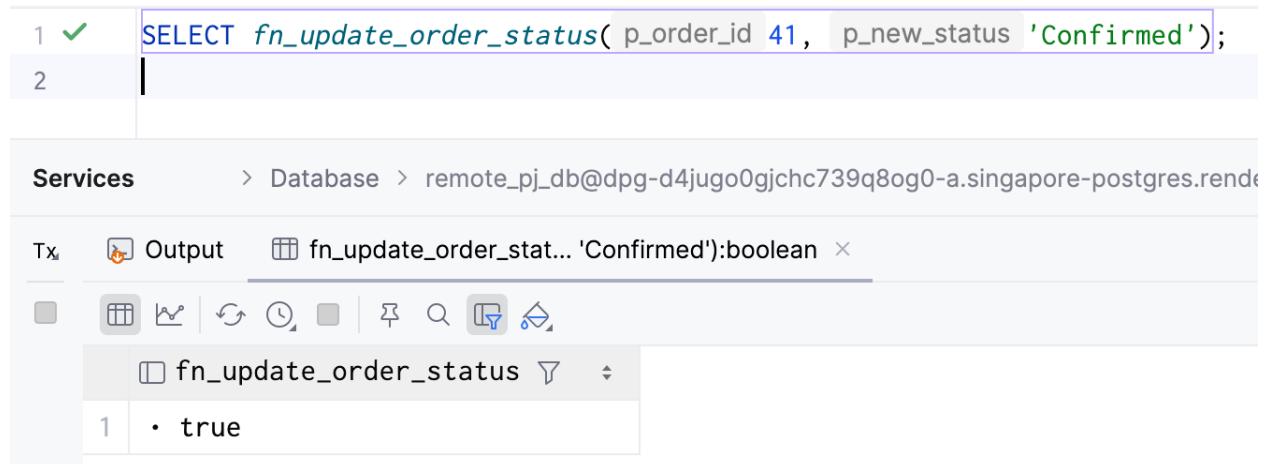
- **Code:**

```
create function fn_update_order_status(p_order_id integer, p_new_status text)
returns boolean
language plpgsql
as
$$
BEGIN
    UPDATE "order"
    SET status = p_new_status
    WHERE order_id = p_order_id;

    RETURN FOUND;
END;
$$;

alter function fn_update_order_status(integer, text) owner to root;
```

- **Result Structure:** Returns BOOLEAN.



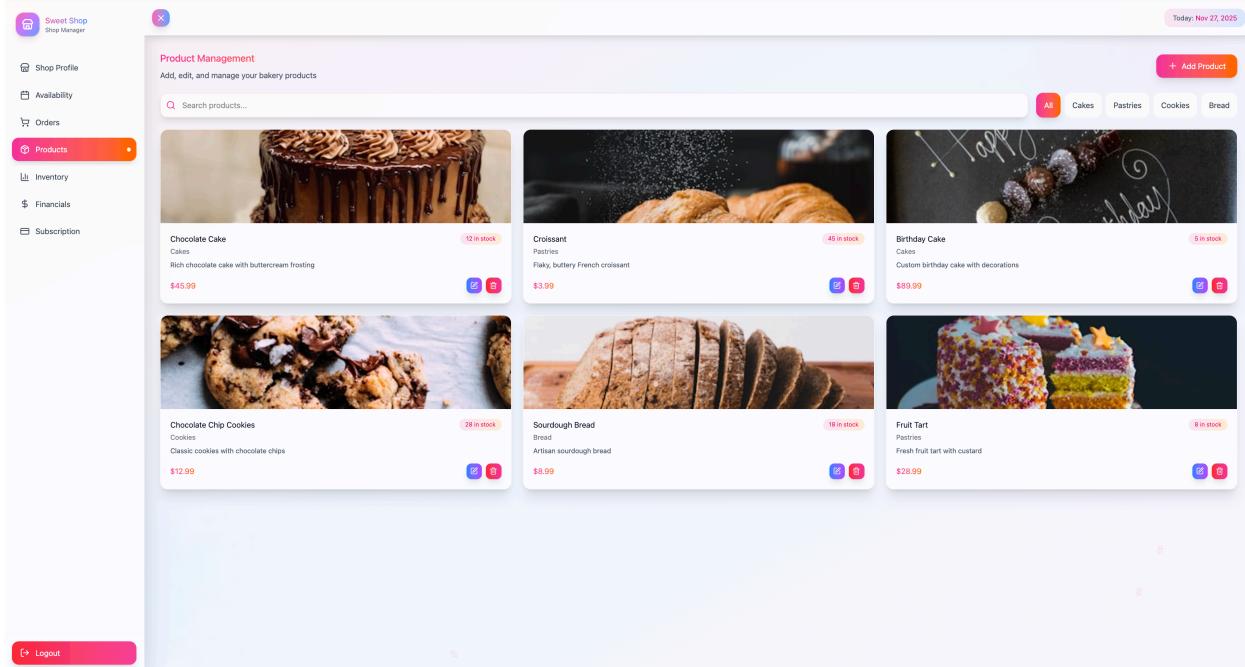
The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓  SELECT fn_update_order_status( p_order_id 41 ,  p_new_status 'Confirmed');
```

The output pane shows the result of the function call:

```
Services          > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com
Tx  Output  fn_update_order_stat... 'Confirmed'):boolean ×
fn_update_order_status  ▾
1  · true
```

## Inventory and Product Catalog



## fn\_get\_shop\_products

- Purpose:** Lists the full product catalog for manager editing, including stock information.
- How It Works (Logic):** Selects detailed product fields filtered by shop\_id. Supports filtering by category and searching by name.
- Code:**

```
create function fn_get_shop_products(p_shop_id integer, p_category_filter text
DEFAULT 'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(product_id integer, name text, description text, price numeric,
category text, stock_quantity integer, is_active boolean, image_url text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.product_id,
            p.name,
            p.description,
            p.price,
            p.category,
            p.stock_quantity,
            p.is_active,
            p.image_url
        FROM
            product p
        WHERE
```

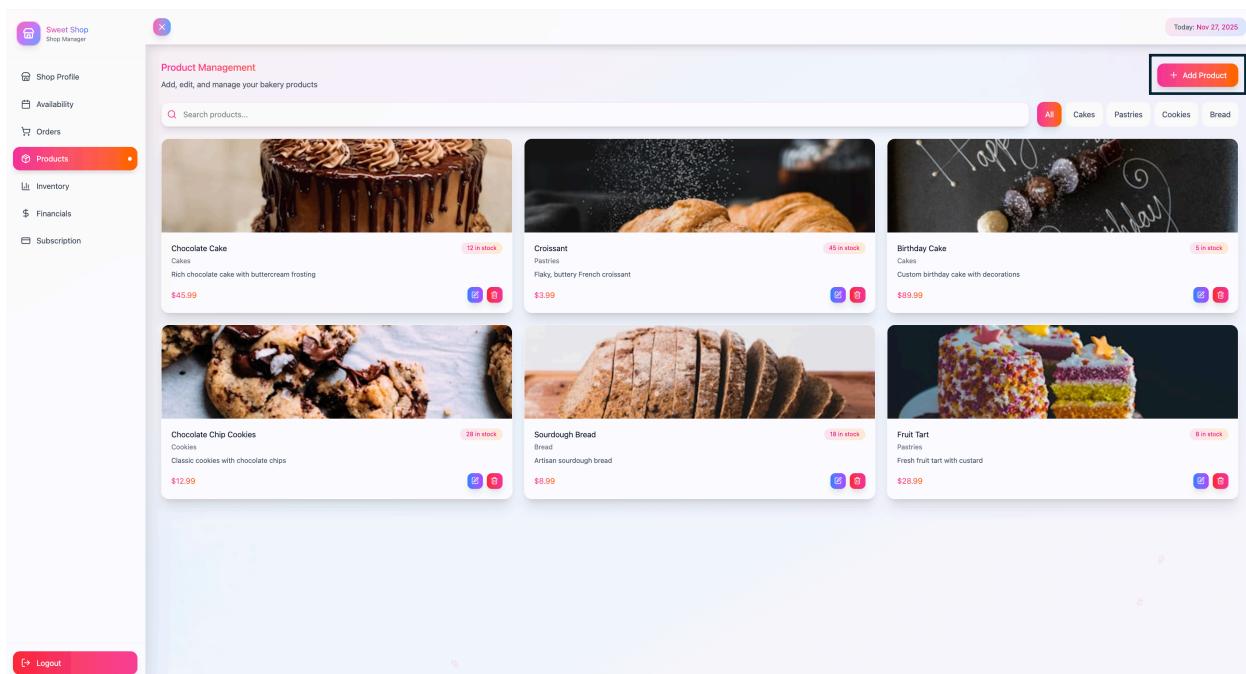
```
p.shop_id = p_shop_id
-- Filter by Category (if not 'All')
AND (p_category_filter = 'All' OR p.category = p_category_filter)
-- Filter by Search (Product Name)
AND (
    p_search_query = ''
    OR p.name ILIKE '%' || p_search_query || '%'
)
ORDER BY
    p.name ASC;
END;
$$;

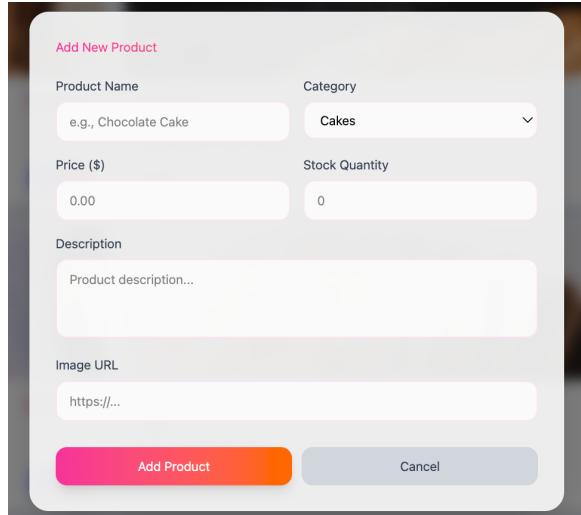
alter function fn_get_shop_products(integer, text, text) owner to root;
```

- **Result Structure:** Returns the management product catalog: (product\_id INTEGER, name TEXT, description TEXT, price NUMERIC, category TEXT, stock\_quantity INTEGER, is\_active BOOLEAN, image\_url TEXT).

```
1 ✓ SELECT * FROM fn_get_shop_products( p_shop_id 20, p_category_filter 'All', p_search_query '' );
2

Services > Database > remote_pj_db@dpq-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console_12
Tx Output Result 46 Result 46-2
i product_id name description price category stock_quant... is_active image_url
1 16 Handmade Candle Lavender scented candle 15.99 Home Decor 20 true https://example.com/candle.jpg
```





## fn\_add\_shop\_product

- Purpose:** Creates a new product entry in the shop's catalog.
- How It Works (Logic):** Inserts a new row into product, defaulting is\_active to TRUE.
- Code:**

```

create function fn_add_shop_product(p_shop_id integer, p_name text, p_description
text, p_price numeric, p_category text, p_stock_quantity integer, p_image_url
text) returns integer
language plpgsql
as
$$
DECLARE
    v_new_id INT;
BEGIN
    INSERT INTO product (
        shop_id, name, description, price, category, stock_quantity, image_url,
        is_active
    )
    VALUES (
        p_shop_id, p_name, p_description, p_price, p_category,
        p_stock_quantity, p_image_url, TRUE
    )
    RETURNING product_id INTO v_new_id;

    RETURN v_new_id;
END;
$$;

alter function fn_add_shop_product(integer, text, text, numeric, text, integer,
text) owner to root;

```

- Result Structure:** Returns the new product's ID: INTEGER.

```

1 ✓ SELECT fn_add_shop_product(
2     p_shop_id 20, p_name 'Red Velvet Cake', p_description 'Moist red velvet', p_price 35.00, p_category 'Cakes', p_stock_quantity 10, p_image_url 'redvelvet.jpg'
3 );
4

```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjhc739q8og0-a.singapore-postgres.render.com > console\_12

Tx Output fn\_add\_shop\_product(....jpg') :integer Result 46-2

fn\_add\_shop\_product :

|   |    |
|---|----|
| 1 | 18 |
|---|----|

CSV ▾

Sweet Shop Shop Manager

Product Management

Add, edit, and manage your bakery products

Search products...

Today: Nov 27, 2023

+ Add Product

Shop Profile Availability Orders Products Inventory Financials Subscription

Chocolate Cake  
Cakes  
Rich chocolate cake with buttercream frosting  
\$45.99

Croissant  
Pastries  
Flaky, buttery French croissant  
\$3.99

Birthday Cake  
Cakes  
Custom birthday cake with decorations  
\$89.99

Chocolate Chip Cookies  
Cookies  
Classic cookies with chocolate chips  
\$12.99

Sourdough Bread  
Bread  
Artisan sourdough bread  
\$8.99

Fruit Tart  
Pastries  
Fresh fruit tart with custard  
\$28.99

Logout

Edit Product

|                                                                                                                                                     |                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| Product Name                                                                                                                                        | Category       |
| Chocolate Cake                                                                                                                                      | Cakes          |
| Price (\$)                                                                                                                                          | Stock Quantity |
| 45.99                                                                                                                                               | 12             |
| Description                                                                                                                                         |                |
| Rich chocolate cake with buttercream frosting                                                                                                       |                |
| Image URL                                                                                                                                           |                |
| <a href="https://images.unsplash.com/photo-1578985545062-69928b1d9587?w=400">https://images.unsplash.com/photo-1578985545062-69928b1d9587?w=400</a> |                |
| <input type="button" value="Update Product"/> <input type="button" value="Cancel"/>                                                                 |                |

## fn\_update\_shop\_product

- Purpose:** Modifies details of an existing product.

- **How It Works (Logic):** Updates fields in product, setting updated\_at = NOW(). Requires product\_id and shop\_id match.

- **Code:**

```

create function fn_update_shop_product(p_product_id integer, p_shop_id integer,
p_name text, p_description text, p_price numeric, p_category text,
p_stock_quantity integer, p_image_url text) returns boolean
    language plpgsql
as
$$
BEGIN
    UPDATE product
    SET
        name = p_name,
        description = p_description,
        price = p_price,
        category = p_category,
        stock_quantity = p_stock_quantity,
        image_url = p_image_url,
        updated_at = NOW()
    WHERE
        product_id = p_product_id
        AND shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

alter function fn_update_shop_product(integer, integer, text, text, numeric, text,
integer, text) owner to root;

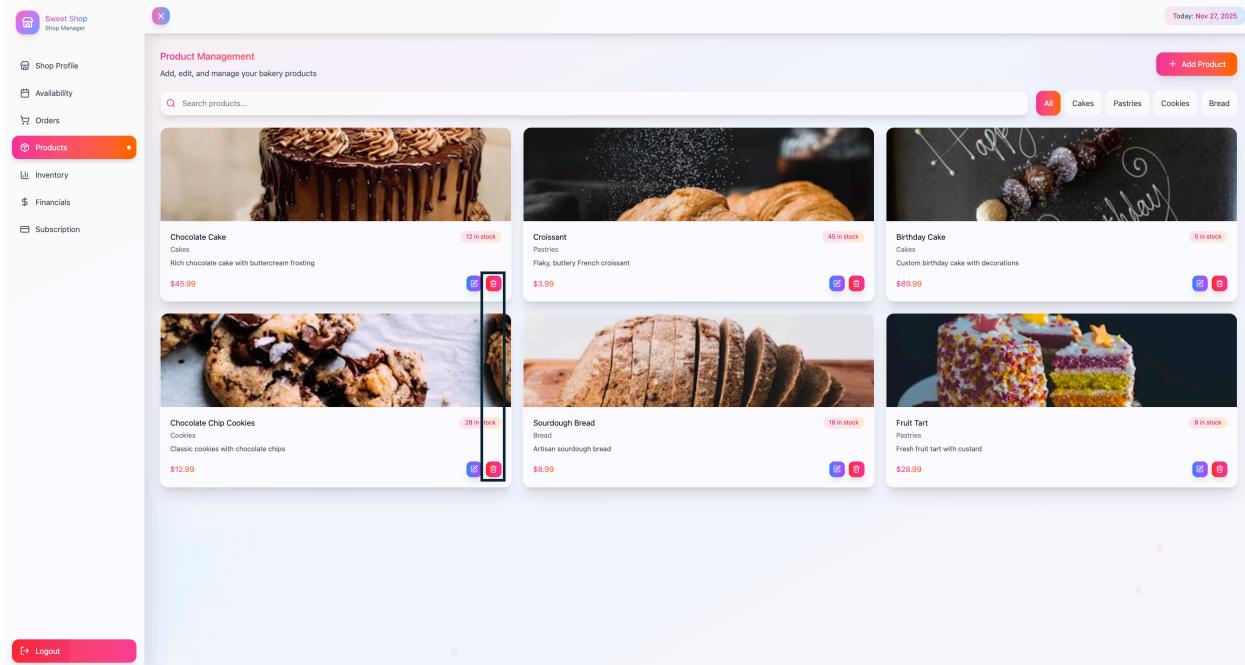
```

- **Result Structure:** Returns BOOLEAN.

```

1 ✓ SELECT fn_update_shop_product(
2     p_product_id 18, p_shop_id 20, p_name 'Red Velvet Cake', p_description 'Updated desc', p_price 32.00, p_category 'Cakes', p_stock_quantity 15, p_image_url 'new.jpg'
3 );
4
Services > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console_12
Tx Output fn_add_shop_product(...).jpg ):integer fn_update_shop_product(...).boolean ×
fn_update_shop_product
1 true
CSV ▾ ↴ ↵

```



## fn\_delete\_shop\_product

- Purpose:** Permanently removes a product from the catalog.
- How It Works (Logic):** Executes DELETE on the product table.
- Code:**

```
create function fn_delete_shop_product(p_product_id integer, p_shop_id integer)
returns boolean
language plpgsql
as
$$
BEGIN
    DELETE FROM product
    WHERE
        product_id = p_product_id
        AND shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

alter function fn_delete_shop_product(integer, integer) owner to root;
```

- Result Structure:** Returns BOOLEAN.

```

1 ✓ SELECT fn_delete_shop_product( p_product_id 18, p_shop_id 20);
2

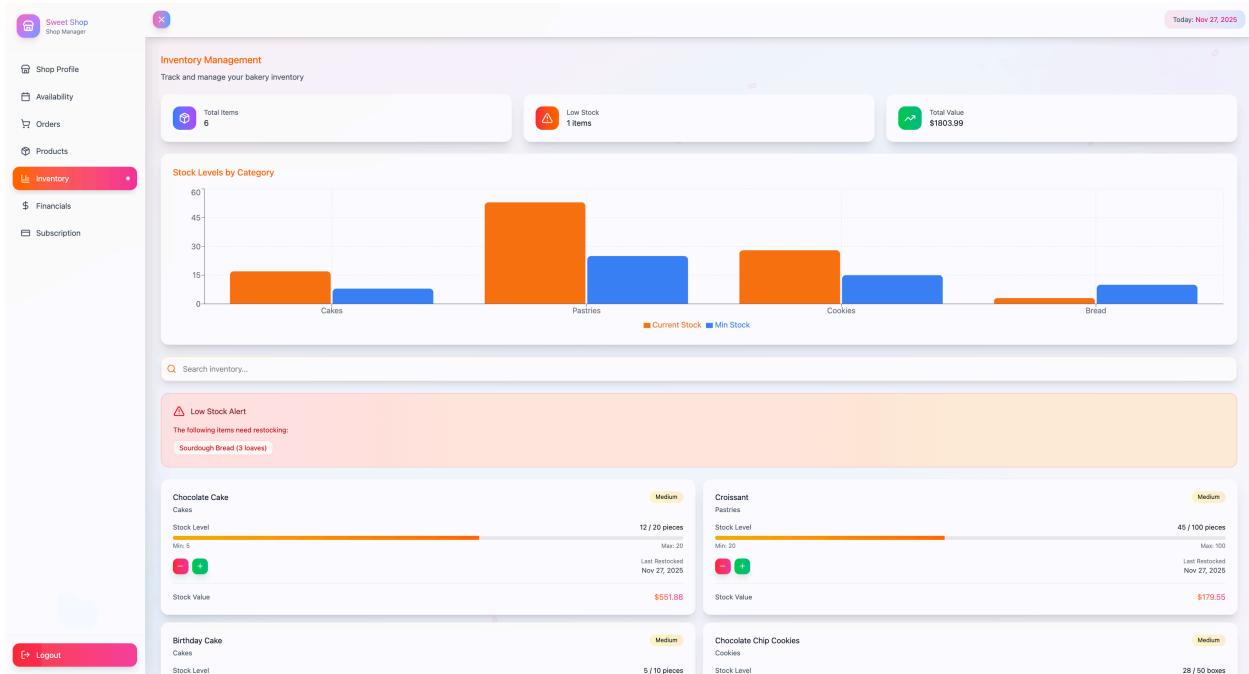
```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-pool-1.emea-central-1.amazonaws.com

Tx Output fn\_delete\_shop\_product(18, 20):boolean fn\_update\_shop\_product(18, 20):void

fn\_delete\_shop\_product

1 true



## fn\_get\_inventory\_stats

- Purpose:** Provides summary metrics for overall inventory health.
- How It Works (Logic):** Aggregates data: COUNT(\*) for total items, COUNT(\*) FILTER for low stock items, and SUM(stock\_quantity \* price) for total\_value.
- Code:**

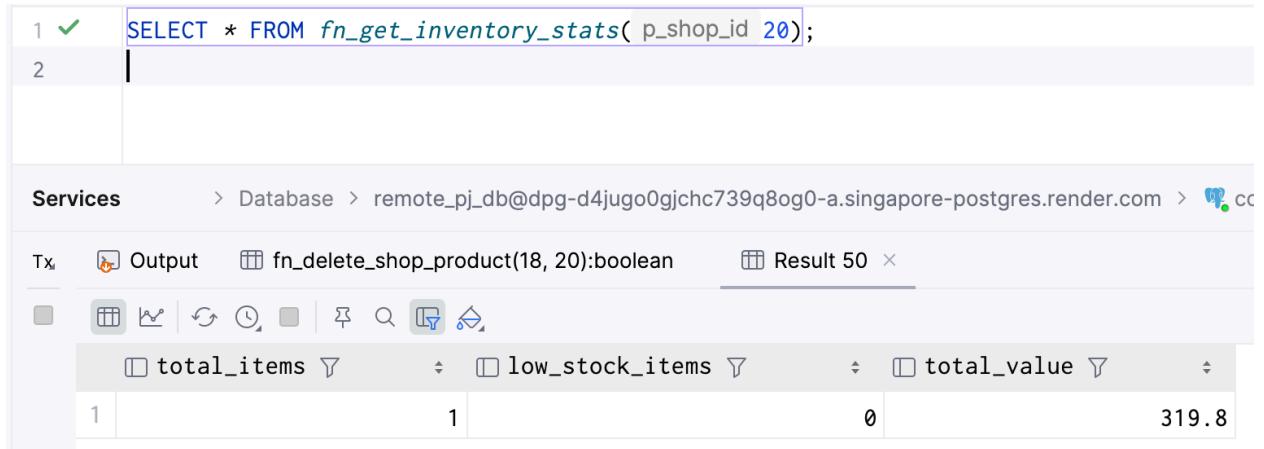
```

create function fn_get_inventory_stats(p_shop_id integer)
    returns TABLE(total_items bigint, low_stock_items bigint, total_value numeric)
language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            COUNT(*)::BIGINT AS total_items,
            COUNT(*) FILTER (WHERE stock_quantity <= min_stock_level)::BIGINT AS
low_stock_items,
            COALESCE(SUM(stock_quantity * price), 0.00)::DECIMAL(10,2) AS
total_value
        FROM
            product
        WHERE
            shop_id = p_shop_id
            AND is_active = TRUE;
END;
$$;

alter function fn_get_inventory_stats(integer) owner to root;

```

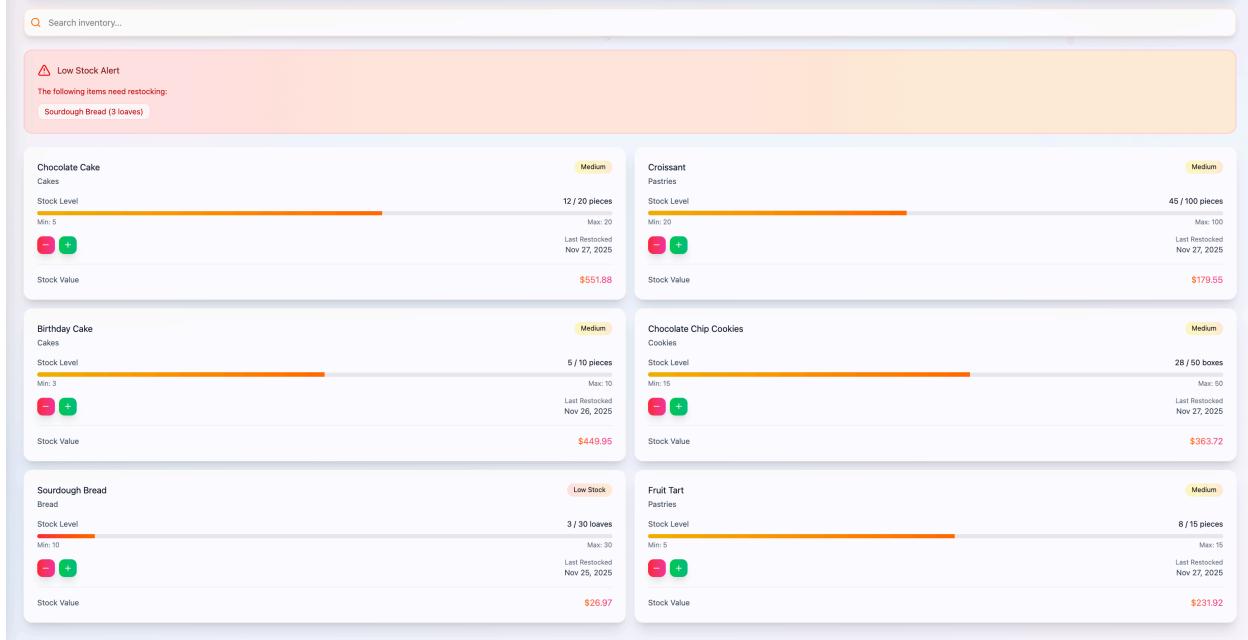
- **Result Structure:** Returns aggregate counts: (total\_items BIGINT, low\_stock\_items BIGINT, total\_value NUMERIC).



The screenshot shows a PostgreSQL database interface with the following details:

- Query Editor:** Contains two numbered lines of SQL code:
  - Line 1: `1 ✓ SELECT * FROM fn_get_inventory_stats( p_shop_id 20);`
  - Line 2: `2 |`
- Services:** Shows the connection path: Services > Database > remote\_pj\_db@dp... > Database.
- Toolbar:** Includes Tx, Output, fn\_delete\_shop\_product(18, 20):boolean, and Result 50.
- Result Grid:** Displays the output of the query:
 

|   | total_items | low_stock_items | total_value |
|---|-------------|-----------------|-------------|
| 1 | 1           | 0               | 319.8       |



## fn\_get\_inventory\_list

- Purpose:** Lists inventory items for stock management, highlighting stock status and value.
- How It Works (Logic):** Selects product data, calculates stock\_value, and uses the generated stock\_status column. Orders results to prioritize low-stock items first.
- Code:**

```
create function fn_get_inventory_list(p_shop_id integer, p_search_query text
DEFAULT ''::text)
    returns TABLE(product_id integer, name text, category text, stock_quantity
integer, min_stock integer, max_stock integer, unit_type text, stock_value
numeric, last_restocked date, status text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.product_id,
            p.name,
            p.category,
            p.stock_quantity,
            p.min_stock_level,
            p.max_stock_level,
            p.unit_type,
            (p.stock_quantity * p.price)::DECIMAL(10,2) AS stock_value,
            p.last_restocked_at::DATE,
            p.stock_status -- Uses the generated column we added

```

```

FROM
    product p
WHERE
    p.shop_id = p_shop_id
    AND p.is_active = TRUE
    AND (
        p_search_query = ''
        OR p.name ILIKE '%' || p_search_query || '%'
        OR p.category ILIKE '%' || p_search_query || '%'
    )
ORDER BY
    -- Prioritize Low Stock items first
    (p.stock_quantity <= p.min_stock_level) DESC,
    p.name ASC;
END;
$$;

alter function fn_get_inventory_list(integer, text) owner to root;

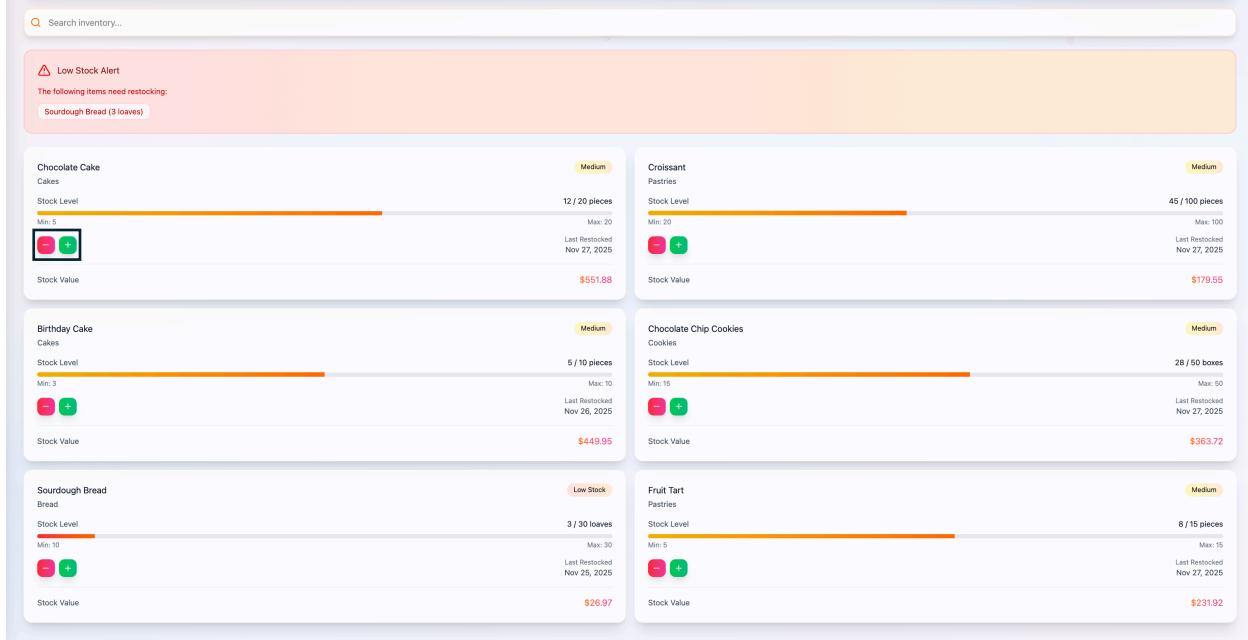
```

- **Result Structure:** Returns detailed inventory list: (product\_id INTEGER, name TEXT, category TEXT, stock\_quantity INTEGER, min\_stock INTEGER, max\_stock INTEGER, unit\_type TEXT, stock\_value NUMERIC, last\_restocked DATE, status TEXT).

The screenshot shows a PostgreSQL database interface with the following details:

- Query:** SELECT \* FROM fn\_get\_inventory\_list( p\_shop\_id 20, p\_search\_query '' );
- Environment:** Services > Database > remote\_pj\_db@dpq-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12
- Output:** Result 51 x Result 50
- Table Headers:** product\_id, name, category, stock\_quantity, min\_stock, max\_stock, unit\_type, stock\_value, last\_restocked
- Table Data:**

|   | product_id | name            | category   | stock_quantity | min_stock | max_stock | unit_type | stock_value | last_restocked |
|---|------------|-----------------|------------|----------------|-----------|-----------|-----------|-------------|----------------|
| 1 | 16         | Handmade Candle | Home Decor | 20             | 5         | 50        | pieces    | 319.8       | 2025-11-26     |



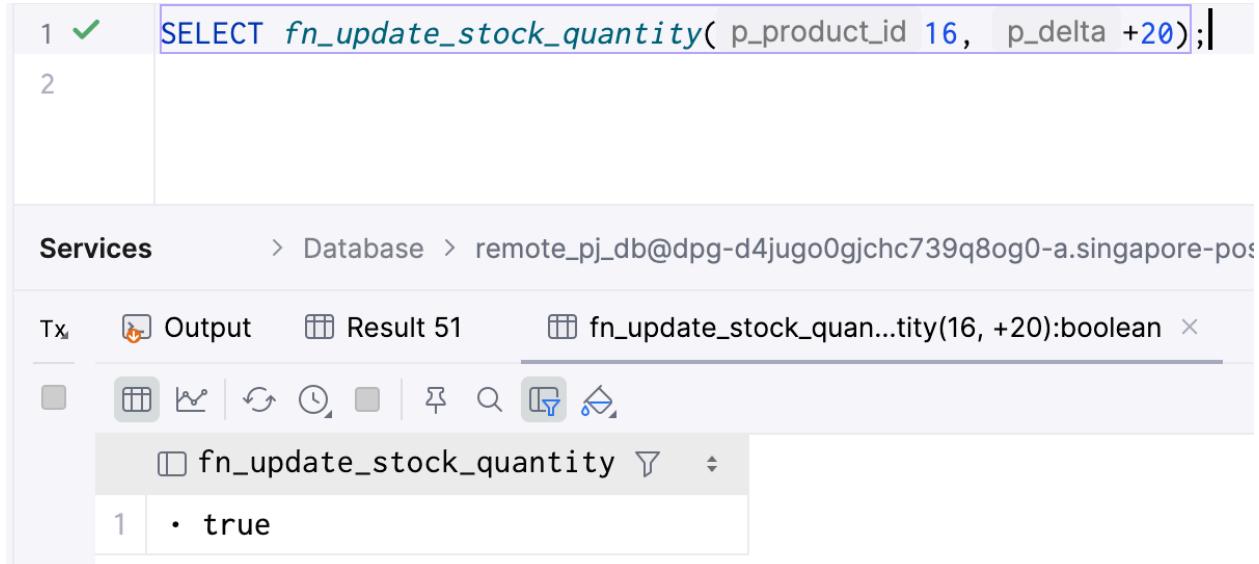
## fn\_update\_stock\_quantity

- Purpose:** Adjusts the stock level of a product.
- How It Works (Logic):** Updates stock\_quantity by adding p\_delta, using GREATEST(0, ...) to prevent negative stock. Updates last\_restocked\_at only if stock was added (p\_delta > 0).
- Code:**
- ```
create function fn_update_stock_quantity(p_product_id integer, p_delta integer)
returns boolean
language plpgsql
as
$$
DECLARE
    v_new_quantity INT;
BEGIN
    -- 1. Update the stock
    UPDATE product
    SET
        stock_quantity = GREATEST(0, stock_quantity + p_delta), -- Prevent
        negative stock
        last_restocked_at = CASE
            WHEN p_delta > 0 THEN NOW() -- Update timestamp
            only if adding stock
            ELSE last_restocked_at
        END
    WHERE product_id = p_product_id
    RETURNING stock_quantity INTO v_new_quantity;
```

```
    RETURN FOUND;
END;
$$;
```

```
alter function fn_update_stock_quantity(integer, integer) owner to root;
```

- **Result Structure:** Returns BOOLEAN.

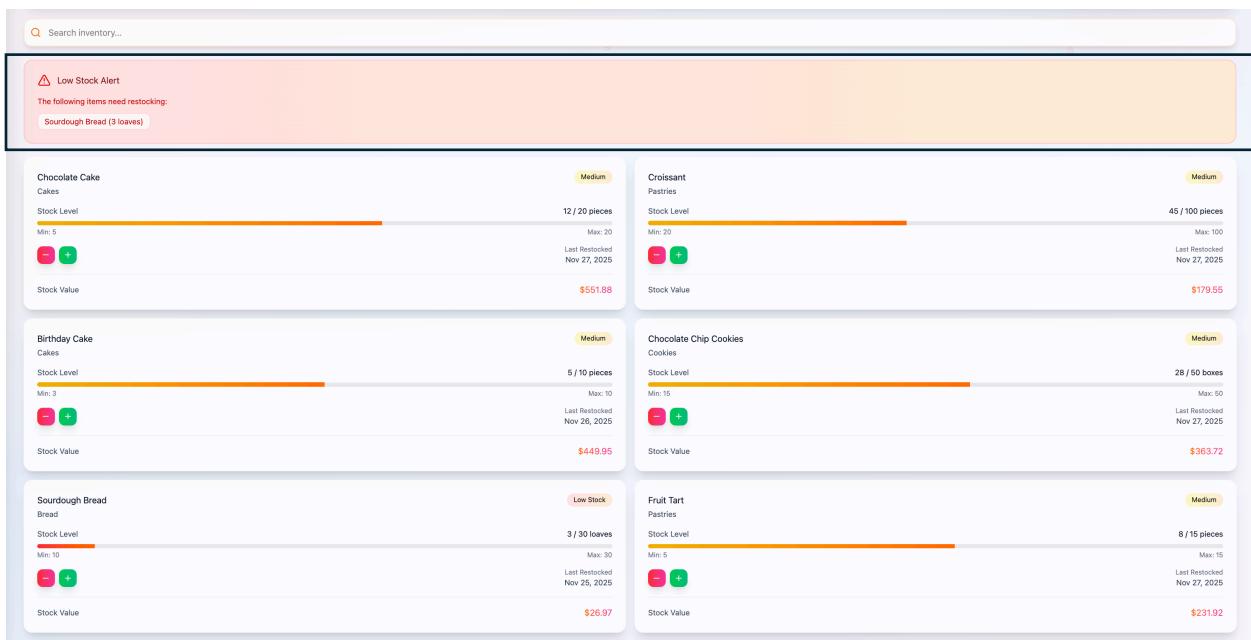


The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓ SELECT fn_update_stock_quantity( p_product_id 16, p_delta +20);|
```

The result set is:

	fn_update_stock_quantity
1	• true



## fn\_get\_low\_stock\_alerts

- **Purpose:** Generates a list of all products currently below their minimum stock threshold.
- **How It Works (Logic):** Filters product where stock\_quantity <= min\_stock\_level.
- **Code:**

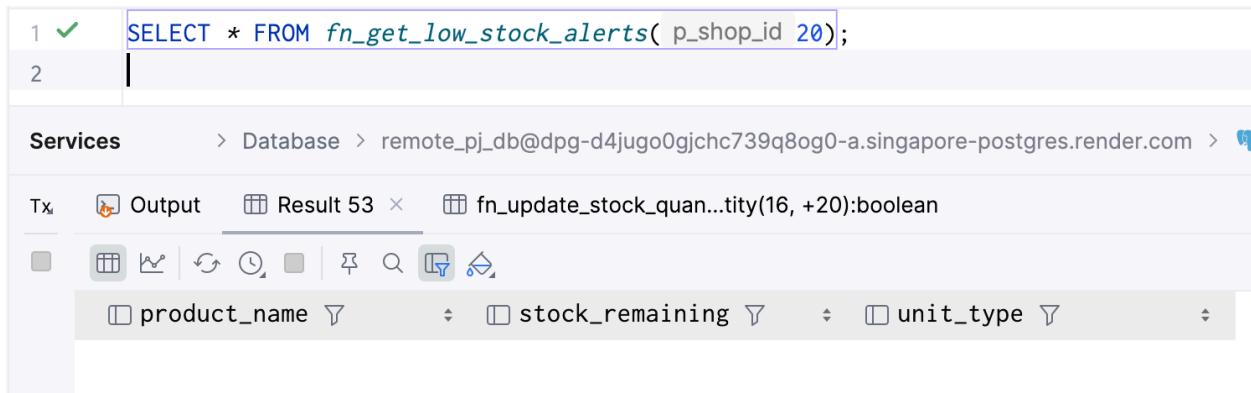
```

create function fn_get_low_stock_alerts(p_shop_id integer)
    returns TABLE(product_name text, stock_remaining integer, unit_type text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.name,
            p.stock_quantity,
            p.unit_type
        FROM
            product p
        WHERE
            p.shop_id = p_shop_id
            AND p.is_active = TRUE
            AND p.stock_quantity <= p.min_stock_level
        ORDER BY
            p.stock_quantity ASC;
END;
$$;

alter function fn_get_low_stock_alerts(integer) owner to root;

```

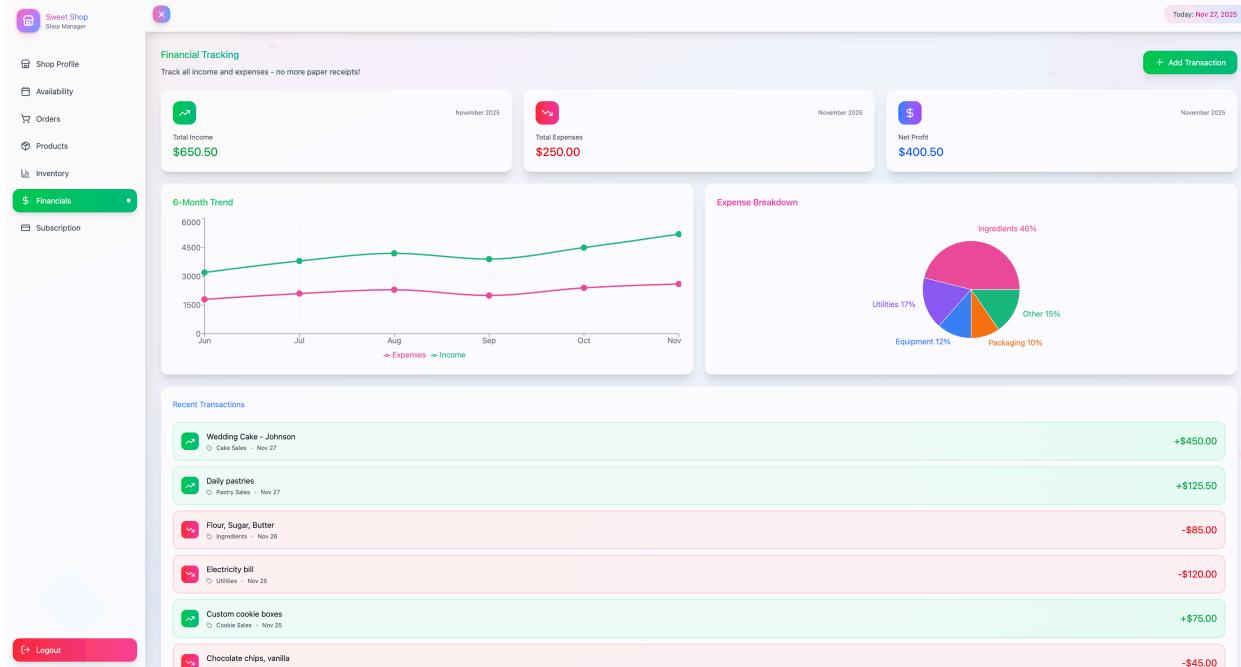
- **Result Structure:** Returns alerts: (product\_name TEXT, stock\_remaining INTEGER, unit\_type TEXT).



The screenshot shows a PostgreSQL database interface with the following details:

- Query Editor:** Contains the SQL command: `SELECT * FROM fn_get_low_stock_alerts( p_shop_id 20);`
- Services:** Shows the connection path: Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com
- Output Tab:** Labeled "Output" and "Result 53".
- Result Table Headers:** The result set includes columns: product\_name, stock\_remaining, and unit\_type.

## Financial Tracking



## fn\_get\_financial\_overview

- Purpose:** Calculates summary financial metrics (Income, Expenses, Net Profit) for a specified month and year.
- How It Works (Logic):** Executes two SUM queries on shop\_transaction, filtered by shop ID, transaction type ('Income'/'Expense'), and month/year. Calculates net\_profit as Income minus Expenses.
- Code:**

```
create function fn_get_financial_overview(p_shop_id integer, p_month integer,
p_year integer)
    returns TABLE(total_income numeric, total_expenses numeric, net_profit
numeric)
    language plpgsql
as
$$
DECLARE
    v_income DECIMAL(10,2);
    v_expenses DECIMAL(10,2);
BEGIN
    -- Calculate Income
    SELECT COALESCE(SUM(amount), 0.00) INTO v_income
    FROM shop_transaction
    WHERE shop_id = p_shop_id
        AND type = 'Income'
        AND EXTRACT(MONTH FROM transaction_date) = p_month
        AND EXTRACT(YEAR FROM transaction_date) = p_year;

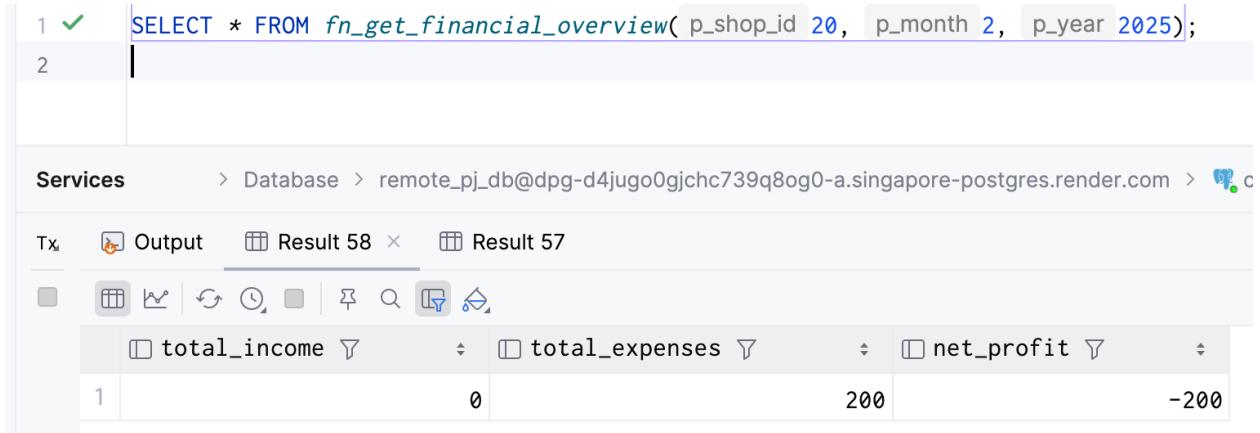
```

```
-- Calculate Expenses
SELECT COALESCE(SUM(amount), 0.00) INTO v_expenses
FROM shop_transaction
WHERE shop_id = p_shop_id
AND type = 'Expense'
AND EXTRACT(MONTH FROM transaction_date) = p_month
AND EXTRACT(YEAR FROM transaction_date) = p_year;

RETURN QUERY
SELECT
    v_income,
    v_expenses,
    (v_income - v_expenses)::DECIMAL(10,2) AS net_profit;
END;
$$;

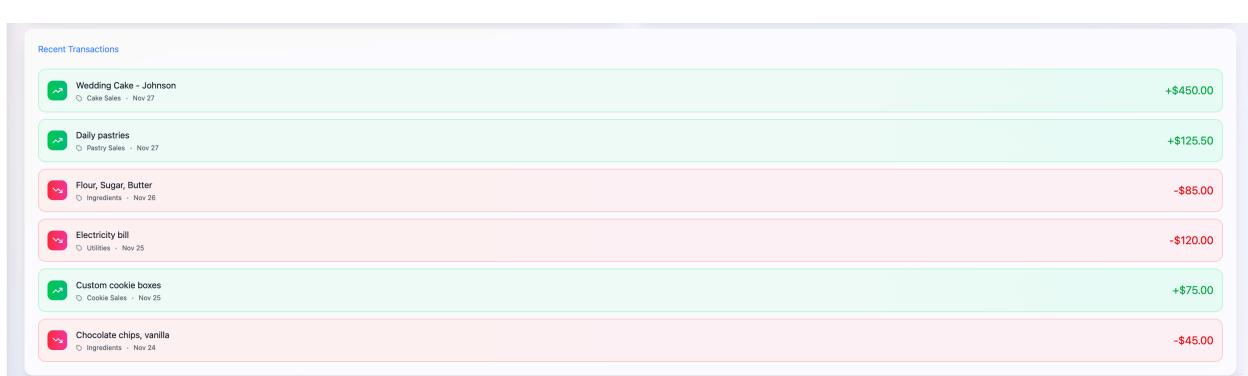
alter function fn_get_financial_overview(integer, integer, integer) owner to root;
```

- Result Structure:** Returns the monthly financial summary: (total\_income NUMERIC, total\_expenses NUMERIC, net\_profit NUMERIC).



The screenshot shows a PostgreSQL database client interface. In the command line, line 1 shows a successful checkmark next to the query, and line 2 shows the start of the query. Below the command line is a results grid. The grid has three columns labeled 'total\_income', 'total\_expenses', and 'net\_profit'. A single row is present with values 0, 200, and -200 respectively.

	total_income	total_expenses	net_profit
1	0	200	-200



## fn\_get\_recent\_transactions

- **Purpose:** Retrieves the shop's most recent financial transactions for a dashboard feed.
- **How It Works (Logic):** Queries shop\_transaction, ordered by date and creation time descending, limited by p\_limit.
- **Code:**

```

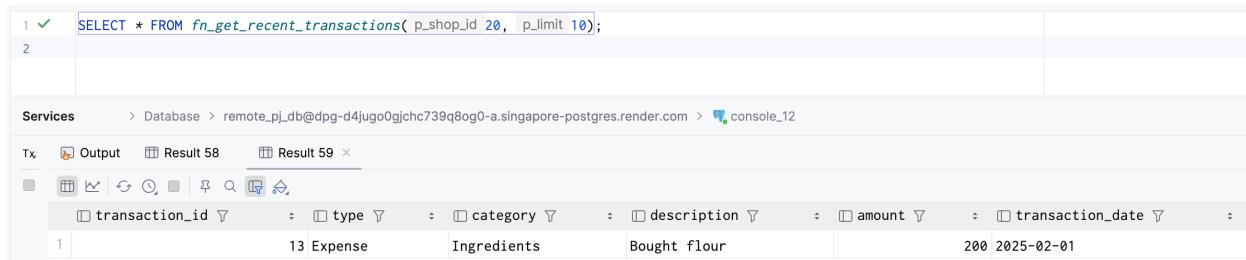
create function fn_get_recent_transactions(p_shop_id integer, p_limit integer
DEFAULT 5)
    returns TABLE(transaction_id integer, type text, category text, description
text, amount numeric, transaction_date date)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            t.transaction_id,
            t.type,
            t.category,
            t.description,
            t.amount,
            t.transaction_date
        FROM
            shop_transaction t
        WHERE
            t.shop_id = p_shop_id
        ORDER BY
            t.transaction_date DESC, t.created_at DESC
        LIMIT p_limit;
END;
$$;

alter function fn_get_recent_transactions(integer, integer) owner to root;

```

- **Result Structure:** Returns transaction feed: (transaction\_id INTEGER, type TEXT, category TEXT, description TEXT, amount NUMERIC, transaction\_date DATE).

1 ✓ SELECT \* FROM fn\_get\_recent\_transactions( p\_shop\_id 20, p\_limit 10);  
2

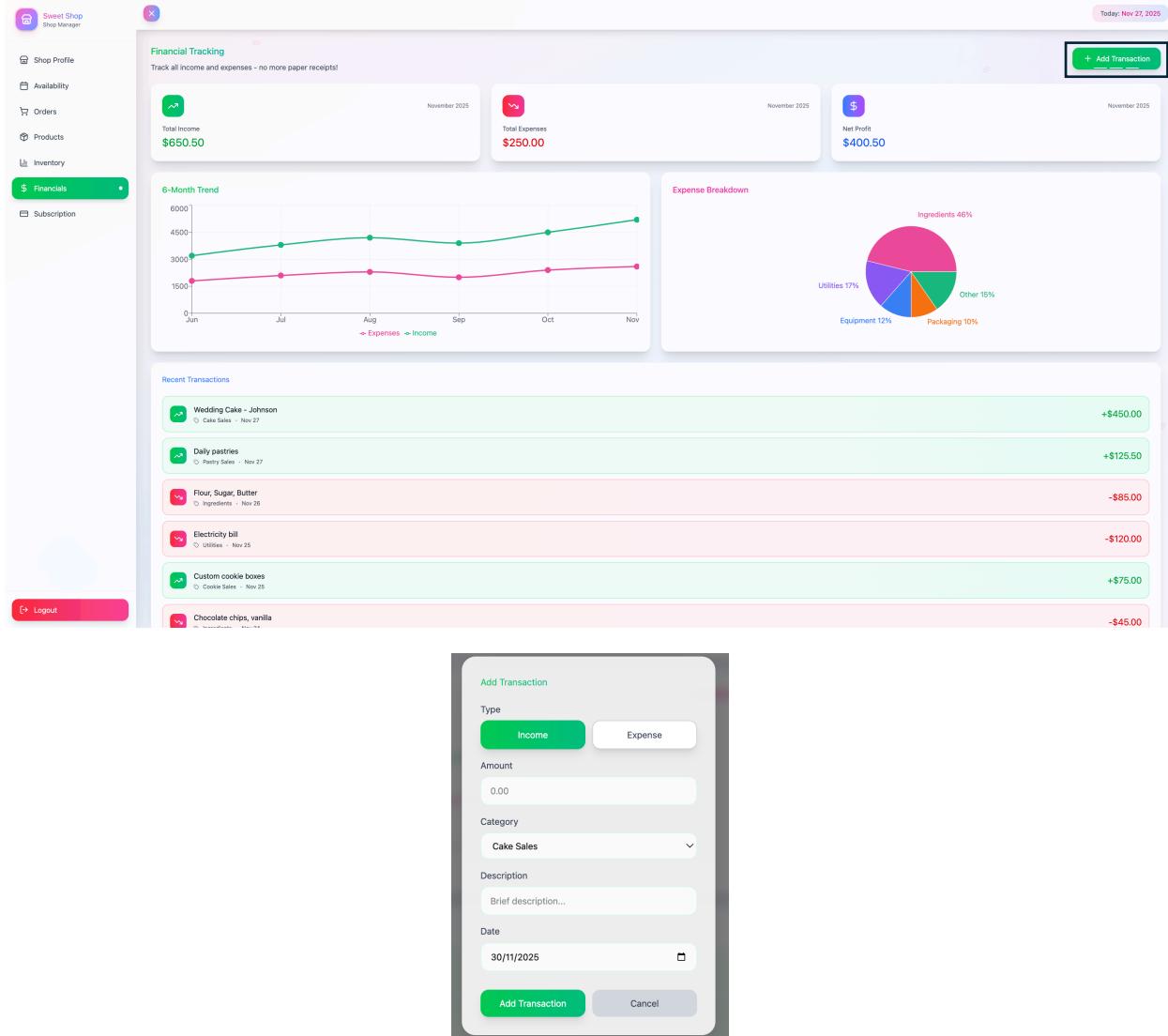


The screenshot shows a PostgreSQL database interface. At the top, there is a command line with the following text:  
1 ✓ SELECT \* FROM fn\_get\_recent\_transactions( p\_shop\_id 20, p\_limit 10);  
2

Below the command line, there is a navigation bar with the following items: Services > Database > remote\_pj\_db@dpq-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12. There are also tabs for Tx, Output, Result 58, and Result 59.

Under the Result 59 tab, there is a table with the following data:

	transaction_id	type	category	description	amount	transaction_date
1	13	Expense	Ingredients	Bought flour	200	2025-02-01



## fn\_add\_transaction

- Purpose:** Inserts a new financial income or expense record into the ledger.
- How It Works (Logic):** Inserts a new row into shop\_transaction.
- Code:**

```
create function fn_add_transaction(p_shop_id integer, p_type text, p_amount
numeric, p_category text, p_description text, p_date date DEFAULT CURRENT_DATE)
returns integer
language plpgsql
as
$$
DECLARE
    v_new_id INT;
BEGIN
    INSERT INTO shop_transaction (
        shop_id, type, amount, category, description, date
    ) VALUES (
        p_shop_id, p_type, p_amount, p_category, p_description, p_date
    );
    SELECT last_value INTO v_new_id FROM shop_transaction
        WHERE column_name = 'id';
    RETURN v_new_id;
END
$$;
```

```

        shop_id, type, amount, category, description, transaction_date
    )
VALUES (
        p_shop_id, p_type, p_amount, p_category, p_description, p_date
    )
RETURNING transaction_id INTO v_new_id;

    RETURN v_new_id;
END;
$$;

alter function fn_add_transaction(integer, text, numeric, text, text, date) owner
to root;

```

- **Result Structure:** Returns the new transaction ID: INTEGER.

```

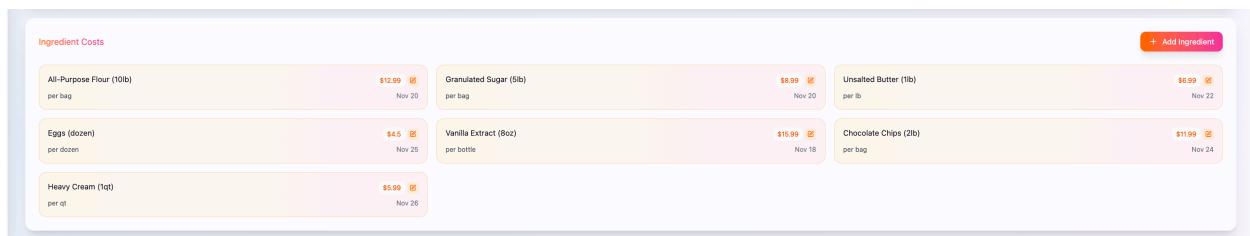
1 ✓  SELECT fn_add_transaction(20, 'Expense', 200.00, 'Ingredients', 'Bought flour', '2025-02-01');
2

```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12

Tx Output fn\_add\_transaction(20, 'Expense', 200.00, 'Ingredients', 'Bought flour', '2025-02-01'):integer Result 54

1	fn_add_transaction	13



## fn\_get\_shop\_ingredients

- **Purpose:** Retrieves the list of raw materials for COGS tracking.
- **How It Works (Logic):** Selects ingredient data filtered by shop\_id.
- **Code:**

```

create function fn_get_shop_ingredients(p_shop_id integer)
    returns TABLE(ingredient_id integer, name text, cost numeric, unit text,
    last_purchased_date date)
    language plpgsql
as
$$
BEGIN

```

```

RETURN QUERY
SELECT
    i.ingredient_id,
    i.name,
    i.cost,
    i.unit,
    i.last_purchased_date
FROM
    shop_ingredient i
WHERE
    i.shop_id = p_shop_id
ORDER BY
    i.name ASC;
END;
$$;

alter function fn_get_shop_ingredients(integer) owner to root;

```

- **Result Structure:** Returns ingredient list: (ingredient\_id INTEGER, name TEXT, cost NUMERIC, unit TEXT, last\_purchased\_date DATE).

```

1 ✓ SELECT * FROM fn_get_shop_ingredients( p_shop_id 20);
2

```

Services > Database > remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console\_12

Tx Output Result 60 × Result 59

	ingredient_id	name	cost	unit	last_purchased_date
1	14	Lavender Oil	5.25	bottle	2025-11-22
2	13	Wax	10.5	kg	2025-11-19

Ingredient	Quantity	Price (\$)	Purchase Date
All-Purpose Flour (10lb)	per bag	\$12.99	Nov 20
Granulated Sugar (5lb)	per bag	\$8.99	Nov 20
Unsalted Butter (1lb)	per lb	\$8.99	Nov 22
Eggs (dozen)	per dozen	\$4.5	Nov 25
Vanilla Extract (8oz)	per bottle	\$15.99	Nov 18
Chocolate Chips (2lb)	per bag	\$11.99	Nov 24

The screenshot shows a PostgreSQL pgAdmin interface with a table named 'Ingredient Costs'. The table has columns: 'Ingredient Name', 'Cost', 'Last Purchased', and a 'Save' button. Below the table, there are five rows of ingredient data:

- All-Purpose Flour (10lb) per bag Cost: \$12.99 Last Purchased: Nov 20
- Granulated Sugar (5lb) per bag Cost: \$8.99 Last Purchased: Nov 20
- Unsalted Butter (1lb) lb Cost: 6.99 Last Purchased: 22/11/2025
- Eggs (dozen) per dozen Cost: \$4.50 Last Purchased: Nov 25
- Vanilla Extract (8oz) per bottle Cost: \$15.99 Last Purchased: Nov 18
- Chocolate Chips (2lb) per bag Cost: \$11.99 Last Purchased: Nov 24
- Heavy Cream (1qt) per qt Cost: \$5.99 Last Purchased: Nov 26

## fn\_upsert\_ingredient

- Purpose:** Creates a new ingredient record or updates an existing one (UPSERT logic).
- How It Works (Logic):** If p\_ingredient\_id is NULL, an INSERT is performed; otherwise, an UPDATE is performed using the provided ID.
- Code:**

```

create function fn_upsert_ingredient(p_shop_id integer, p_ingredient_id integer,
p_name text, p_cost numeric, p_unit text, p_purchase_date date) returns integer
language plpgsql
as
$$
DECLARE
    v_id INT;
BEGIN
    IF p_ingredient_id IS NULL THEN
        -- Create
        INSERT INTO shop_ingredient (shop_id, name, cost, unit,
last_purchased_date)
        VALUES (p_shop_id, p_name, p_cost, p_unit, p_purchase_date)
        RETURNING ingredient_id INTO v_id;
    ELSE
        -- Update
        UPDATE shop_ingredient
        SET
            name = p_name,
            cost = p_cost,
            unit = p_unit,
            last_purchased_date = p_purchase_date
        WHERE
            ingredient_id = p_ingredient_id
            AND shop_id = p_shop_id;
        v_id := p_ingredient_id;
    END IF;

    RETURN v_id;

```

```

END;
$$;

alter function fn_upsert_ingredient(integer, integer, text, numeric, text, date)
owner to root;

```

- **Result Structure:** Returns the ID of the ingredient: INTEGER.

```

1 ✓ SELECT fn_upsert_ingredient( p_shop_id 20, p_ingredient_id NULL, p_name 'Flour', p_cost 25.00, p_unit 'kg', p_purchase_date '2025-02-01');
2 ✓ SELECT fn_upsert_ingredient( p_shop_id 20, p_ingredient_id 5, p_name 'Flour Premium', p_cost 30.00, p_unit 'kg', p_purchase_date '2025-02-01');

Services > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > 📈 console_12
Tx Output fn_upsert_ingredient...'2025-02-01'):integer 2 × fn_upsert_ingredient...'2025-02-01'):integer
fn_upsert_ingredient
1 5

```

Ingredient	Unit	Cost	Purchase Date
All-Purpose Flour (10lb)	per bag	\$12.99	Nov 20
Granulated Sugar (5lb)	per bag	\$8.99	Nov 20
Unsalted Butter (1lb)	lb	6.99	22/11/2025
Eggs (dozen)	per dozen	\$4.5	Nov 25
Vanilla Extract (8oz)	per bottle	\$10.99	Nov 18
Chocolate Chips (2lb)	per bag	\$11.99	Nov 24
Heavy Cream (1qt)	per qt	\$5.99	Nov 26

## fn\_delete\_ingredient

- **Purpose:** Removes an ingredient record.
- **How It Works (Logic):** Executes DELETE on shop\_ingredient.
- **Code:**

```

create function fn_delete_ingredient(p_ingredient_id integer, p_shop_id integer)
returns boolean
language plpgsql
as
$$
BEGIN
    DELETE FROM shop_ingredient
    WHERE
        ingredient_id = p_ingredient_id
        AND shop_id = p_shop_id;

    RETURN FOUND;

```

```
END;  
$$;  
  
alter function fn_delete_ingredient(integer, integer) owner to root;
```

- **Result Structure:** Returns BOOLEAN.



The screenshot shows a PostgreSQL database client interface. At the top, there is a code editor window with two lines of SQL:

```
1 ✓ SELECT fn_delete_ingredient( p_ingredient_id 16, p_shop_id 20);  
2 |
```

Below the code editor is a navigation bar with "Services" and "Database" selected, followed by the connection details "remote\_pj\_db@dpg-d4jugo0gjchc739q8og0-a.singapore".

The main area displays the results of the query:

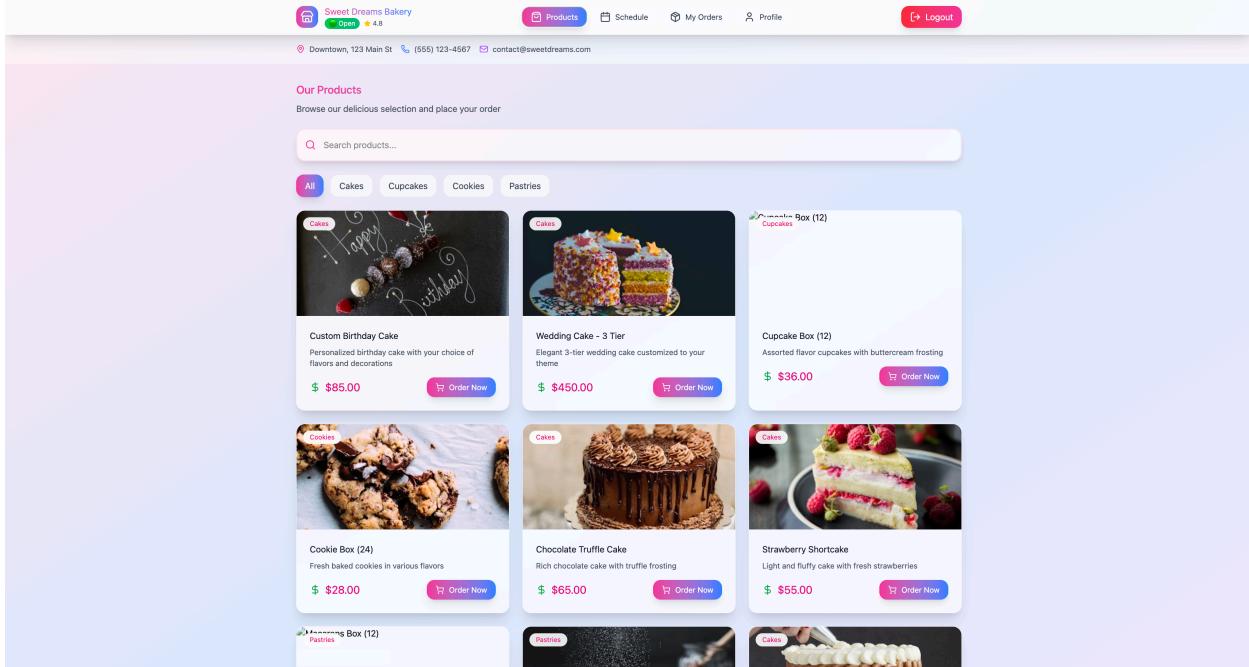
Tx Output fn\_delete\_ingredient(16, 20):boolean

fn\_delete\_ingredient

1	• true

The result shows a single row with a value of "true".

### c. Customer-Facing Functions



## fn\_get\_customer\_products

- Purpose:** Lists products available to customers on the public store front.
- How It Works (Logic):** Filters the product catalog by `is_active = TRUE` and explicitly enforces `stock_quantity > 0` (only showing items available to buy). Supports category and search filtering.
- Code:**

```

create function fn_get_customer_products(p_shop_id integer, p_category_filter text
DEFAULT 'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(product_id integer, name text, description text, price numeric,
category text, image_url text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.product_id,
            p.name,
            p.description,
            p.price,
            p.category,
            p.image_url
        FROM
            product p
        WHERE
            p.shop_id = p_shop_id

```

```

        AND p.is_active = TRUE
        AND p.stock_quantity > 0 -- Only show in-stock items
        AND (p_category_filter = 'All' OR p.category = p_category_filter)
        AND (
            p_search_query = ''
            OR p.name ILIKE '%' || p_search_query || '%'
        )
    ORDER BY
        p.category, p.name;
END;
$$;

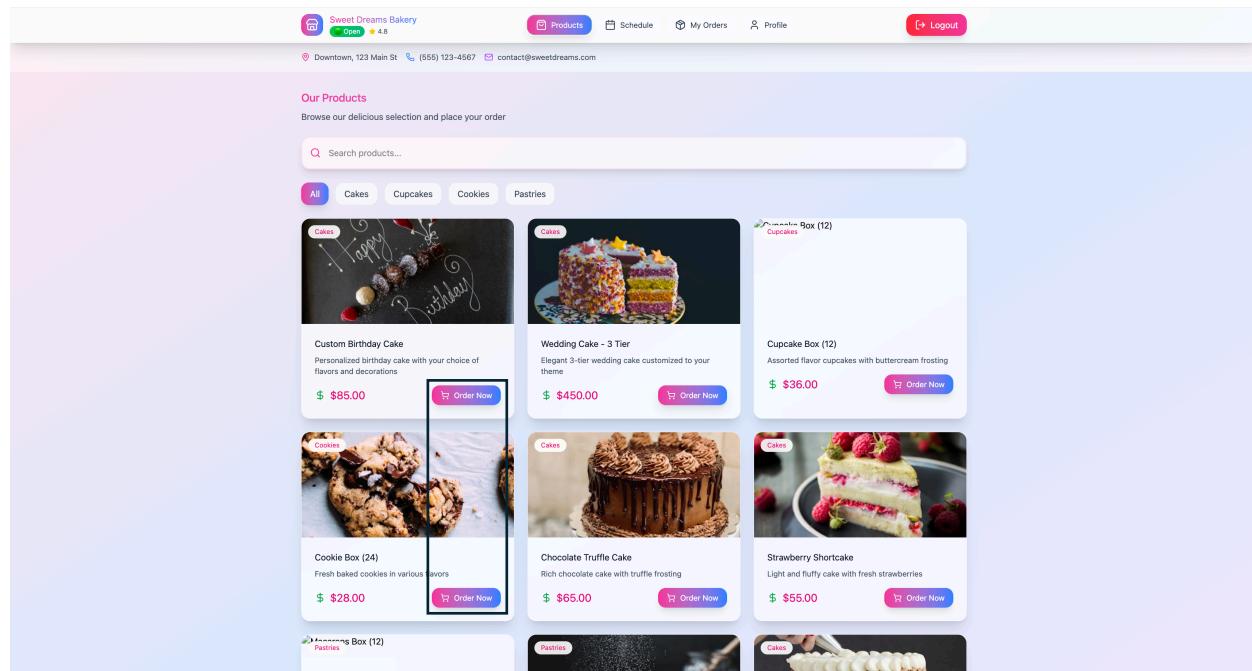
alter function fn_get_customer_products(integer, text, text) owner to root;

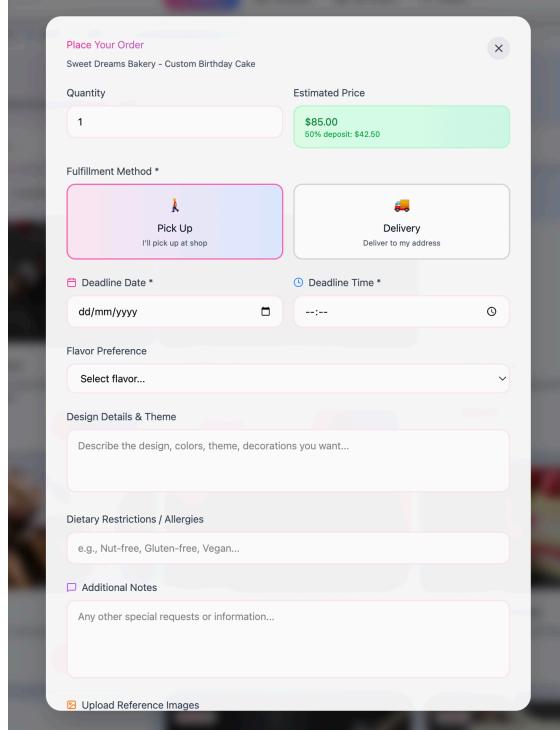
```

- **Result Structure:** Returns the public product catalog: (product\_id INTEGER, name TEXT, description TEXT, price NUMERIC, category TEXT, image\_url TEXT).

The screenshot shows a PostgreSQL query editor interface. The top part contains the SQL code for the `fn_get_customer_products` function. The bottom part shows the results of a query executed against this function, returning 75 rows of product data.

product_id	name	description	price	category	image_url
1	Custom Birthday Cake	Personalized birthday cake with your choice of flavors and decorations	\$85.00	Cakes	
2	Wedding Cake - 3 Tier	Elegant 3-tier wedding cake customized to your theme	\$450.00	Cakes	
3	Cupcake Box (12)	Assorted flavor cupcakes with buttercream frosting	\$36.00	Cupcakes	
4	Cookie Box (24)	Fresh baked cookies in various flavors	\$28.00	Cookies	
5	Chocolate Truffle Cake	Rich chocolate cake with truffle frosting	\$65.00	Cakes	
6	Strawberry Shortcake	Light and fluffy cake with fresh strawberries	\$55.00	Cakes	





### fn\_submit\_customer\_order

- **Purpose:** Handles the initial submission of a custom order request from a customer.
- **How It Works (Logic):** Calculates a preliminary v\_estimated\_price. It creates the main order record, setting the initial status to 'Awaiting Quote'. It then inserts the order\_item details, combining all custom notes (flavor, design, dietary) into a single field.
- **Code:**

```

create function fn_submit_customer_order(p_shop_id integer, p_customer_id integer,
                                         p_product_name text, p_quantity integer, p_deadline date, p_deadline_time time
                                         without time zone, p_fulfillment_type text, p_flavor text, p_design_notes text,
                                         p_dietary_notes text) returns integer
language plpgsql
as
$$
DECLARE
    v_new_order_id INT;
    v_combined_notes TEXT;
    v_estimated_price DECIMAL(10,2);
BEGIN
    SELECT price * p_quantity INTO v_estimated_price
    FROM product
    WHERE shop_id = p_shop_id AND name = p_product_name
    LIMIT 1;

    INSERT INTO "order" (

```

```

shop_id,
customer_id,
status,
total_amount,
deadline,
notes
)
VALUES (
    p_shop_id,
    p_customer_id,
    'Awaiting Quote', -- Initial status
    COALESCE(v_estimated_price, 0.00), -- Placeholder price
    (p_deadline + p_deadline_time), -- Combine Date+Time
    'Fulfillment: ' || p_fulfillment_type
)
RETURNING order_id INTO v_new_order_id;

v_combined_notes := 'Flavor: ' || p_flavor ||
    '. Design: ' || p_design_notes ||
    '. Dietary: ' || p_dietary_notes;

INSERT INTO order_item (
    order_id,
    product_name,
    quantity,
    price,
    notes
)
VALUES (
    v_new_order_id,
    p_product_name,
    p_quantity,
    COALESCE(v_estimated_price, 0.00),
    v_combined_notes
);

RETURN v_new_order_id;
END;
$$;

alter function fn_submit_customer_order(integer, integer, text, integer, date,
time, text, text, text, text) owner to root;

```

- **Result Structure:** Returns the new order ID: INTEGER.

```

1 ✓ SELECT fn_submit_customer_order(
2     p_shop_id 20, p_customer_id 23,
3     p_product_name 'Chocolate Cake', p_quantity 2,
4     p_deadline '2025-02-15', p_deadline_time '14:00',
5     p_fulfillment_type 'Pickup',
6     p_flavor 'Chocolate', p_design_notes 'Birthday theme', p_dietary_notes 'No nuts'
7 );
o

```

Output fn\_submit\_customer\_order():integer ×

fn\_submit\_customer\_order ↴ 1 42

## fn\_get\_customer\_orders

- Purpose:** Lists a customer's personal order history and current active orders.
- How It Works (Logic):** Queries order filtered by customer\_id. Uses a subquery with string\_agg to create a concise product\_summary of items ordered. Flags orders ready for pickup/delivery as is\_ready.
- Code:**

- ```

create function fn_get_customer_orders(p_customer_id integer, p_status_filter text
DEFAULT 'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(order_id integer, product_summary text, status text,
total_amount numeric, deposit_amount numeric, deposit_status text, deadline
timestamp with time zone, created_at timestamp with time zone, is_ready boolean)
        language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            o.order_id,
            COALESCE(
                (SELECT string_agg(quantity || 'x ' || product_name, ', ')
                 FROM order_item oi WHERE oi.order_id = o.order_id),
                'Custom Order'
            )::TEXT AS product_summary,
            o.status,
            o.total_amount,
            o.deposit_amount,
            o.deposit_status,
            o.deadline,
            o.created_at,
            (o.status = 'Ready for Pickup' OR o.status = 'Ready for Delivery') AS
is_ready
        FROM
            "order" o
        WHERE
            o.customer_id = p_customer_id
            AND (p_status_filter = 'All' OR o.status = p_status_filter)
            AND (
                p_search_query = ''
                    OR o.order_id::TEXT ILIKE '%' || p_search_query || '%'
                    OR EXISTS (
                        SELECT 1 FROM order_item oi
                        WHERE oi.order_id = o.order_id
                            AND oi.product_name ILIKE '%' || p_search_query || '%'
                    )
            )
        ORDER BY
            o.created_at DESC;
END;
$$;

```

alter function fn\_get\_customer\_orders(integer, text, text) owner to root;

- Result Structure:** Returns the customer's order list: (order\_id INTEGER, product\_summary TEXT, status TEXT, total\_amount NUMERIC, deposit\_amount NUMERIC, deposit\_status TEXT, deadline TIMESTAMP WITH TIME ZONE, created\_at TIMESTAMP WITH TIME ZONE, is\_ready BOOLEAN).

```

1 ✓ SELECT * FROM fn_get_customer_orders( p_customer_id 23, p_status_filter 'All', p_search_query '');
2

```

Output Result 78 ×

| order_id | product_summary      | status         | total_amount | deposit_amount | deposit_status | deadline               |
|----------|----------------------|----------------|--------------|----------------|----------------|------------------------|
| 1        | 42 2x Chocolate Cake | Awaiting Quote | 0            | 0              | Pending        | 2025-02-15 14:00:00.00 |

Sweet Dreams Bakery Open ★ 4.8

Products Schedule My Orders Profile Logout

Downtown, 123 Main St (555) 123-4567 contact@sweatdreams.com

### My Orders

Track all your bakery orders in one place

**ORD-1001** In Process

1x Custom Birthday Cake  
Ordered: 2025-11-20

Total: \$125.00  
Deposit: \$62.50 (PAID)

Shop is working on your order

**ORD-1002** \$ Awaiting Payment 2 days

2x Cookie Box (24)  
Ordered: 2025-11-29

Total: \$56.00  
Deposit: \$28.00 (UNPAID)

Pay deposit to confirm order

**ORD-1003** Awaiting Quote

1x Wedding Cake - 3 Tier  
Ordered: 2025-11-28

Shop is reviewing your order

**Order Details - ORD-1001**

In Process

**Order Information**

Product: Custom Birthday Cake  
Quantity: 1  
Order Date: 2025-11-20  
Deadline: 2025-12-05 at 2:00 PM  
Fulfillment: Pick Up  
Flavor: Vanilla with strawberry filling  
Design: Two-tier cake with gold accents and fresh flowers  
Notes: Pink and gold theme with flowers

**Payment Information**

|                 |          |
|-----------------|----------|
| Total Price:    | \$125.00 |
| Deposit (50%):  | \$62.50  |
| Deposit Status: | Paid     |

## fn\_get\_customer\_order\_details

- **Purpose:** Fetches detailed information for a single order, strictly ensuring the order belongs to the requesting customer.
- **How It Works (Logic):** Joins order and order\_item. The WHERE clause requires both order\_id and customer\_id match for security.
- **Code:**

```

create function fn_get_customer_order_details(p_order_id integer, p_customer_id
integer)
    returns TABLE(order_id integer, status text, product_name text, quantity
integer, order_date date, deadline timestamp with time zone, fulfillment_method
text, delivery_address text, item_notes text, total_price numeric, deposit_amount
numeric, deposit_status text, payment_slip_url text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            o.order_id,
            o.status,
            oi.product_name,
            oi.quantity,
            o.created_at::DATE AS order_date,
            o.deadline,
            o.fulfillment_method,
            o.delivery_address,
            oi.notes AS item_notes,
            o.total_amount,
            o.deposit_amount,
            o.deposit_status,
            o.payment_slip_url
        FROM
            "order" o
                JOIN
            order_item oi ON o.order_id = oi.order_id
        WHERE
            o.order_id = p_order_id
            AND o.customer_id = p_customer_id
        LIMIT 1;
END;
$$;

alter function fn_get_customer_order_details(integer, integer) owner to root;

```

- **Result Structure:** Returns the full details of one order: (order\_id INTEGER, status TEXT, product\_name TEXT, quantity INTEGER, order\_date DATE, deadline TIMESTAMP WITH TIME ZONE, fulfillment\_method TEXT, delivery\_address TEXT,

item\_notes TEXT, total\_price NUMERIC, deposit\_amount NUMERIC, deposit\_status TEXT, payment\_slip\_url TEXT).

```

1 ✓ SELECT * FROM fn_get_customer_order_details( p_order_id 42, p_customer_id 23);
2

```

|   | order_id | status         | product_name   | quantity | order_date | deadline                          | fulfillment_method |
|---|----------|----------------|----------------|----------|------------|-----------------------------------|--------------------|
| 1 | 42       | Awaiting Quote | Chocolate Cake | 2        | 2025-11-30 | 2025-02-15 14:00:00.000000 +00:00 | <null>             |

The screenshot shows the Sweet Dreams Bakery website's availability page. At the top, there are navigation links for Products, Schedule, My Orders, Profile, and Logout. Below that, it displays the address: Downtown, 123 Main St., (555) 123-4567, contact@sweetdreams.com.

**Availability Schedule:** Check our hours and plan your orders - no need to call!

**Today's Hours - Monday:** Regular hours, 8:00 AM - 6:00 PM

**Weekly Schedule:**

- Monday (Today):** Regular hours, 8:00 AM - 6:00 PM, Available
- Tuesday:** Regular hours, 8:00 AM - 6:00 PM, Available
- Wednesday:** Regular hours, 8:00 AM - 6:00 PM, Available
- Thursday:** Regular hours, 8:00 AM - 6:00 PM, Available
- Friday:** Extended hours, 8:00 AM - 8:00 PM, Available
- Saturday:** Weekend hours, 9:00 AM - 8:00 PM, Available
- Sunday:** Closed for rest, Closed

**Special Dates & Holidays:**

- Christmas Day:** Thursday, December 25, 2025, Closed
- Christmas Eve - Close at 2 PM:** Wednesday, December 24, 2025, Limited
- New Year's Eve - Limited orders only:** Wednesday, December 31, 2025, Busy

## fn\_get\_customer\_weekly\_schedule

- Purpose:** Retrieves and formats the full weekly schedule for the customer-facing front end.
- How It Works (Logic):** Selects all weekly availability data, formats the time range, and determines the `is_today` flag for contextual display. Orders the days chronologically.
- Code:**

```

create function fn_get_customer_weekly_schedule(p_shop_id integer)
    returns TABLE(day_of_week text, time_range text, status_tag text, is_today
boolean)
    language plpgsql
as
$$

```

```

DECLARE
    v_today_name TEXT;
BEGIN
    SELECT TRIM(TO_CHAR(CURRENT_DATE, 'Day')) INTO v_today_name;

    RETURN QUERY
        SELECT
            sa.day_of_week::TEXT,
            CASE
                WHEN sa.is_open THEN
                    TO_CHAR(sa.open_time, 'FMHH12:MI AM') || ' - ' || 
                    TO_CHAR(sa.close_time, 'FMHH12:MI PM')
                ELSE 'Closed'
            END::TEXT AS time_range,
            CASE
                WHEN sa.is_open THEN 'Available'
                ELSE 'Closed'
            END::TEXT AS status_tag,
            (sa.day_of_week = v_today_name) AS is_today
        FROM
            shop_availability sa
        WHERE
            sa.shop_id = p_shop_id
        ORDER BY
            CASE
                WHEN sa.day_of_week = 'Monday' THEN 1
                WHEN sa.day_of_week = 'Tuesday' THEN 2
                WHEN sa.day_of_week = 'Wednesday' THEN 3
                WHEN sa.day_of_week = 'Thursday' THEN 4
                WHEN sa.day_of_week = 'Friday' THEN 5
                WHEN sa.day_of_week = 'Saturday' THEN 6
                WHEN sa.day_of_week = 'Sunday' THEN 7
            END;
END;
$$;

alter function fn_get_customer_weekly_schedule(integer) owner to root;

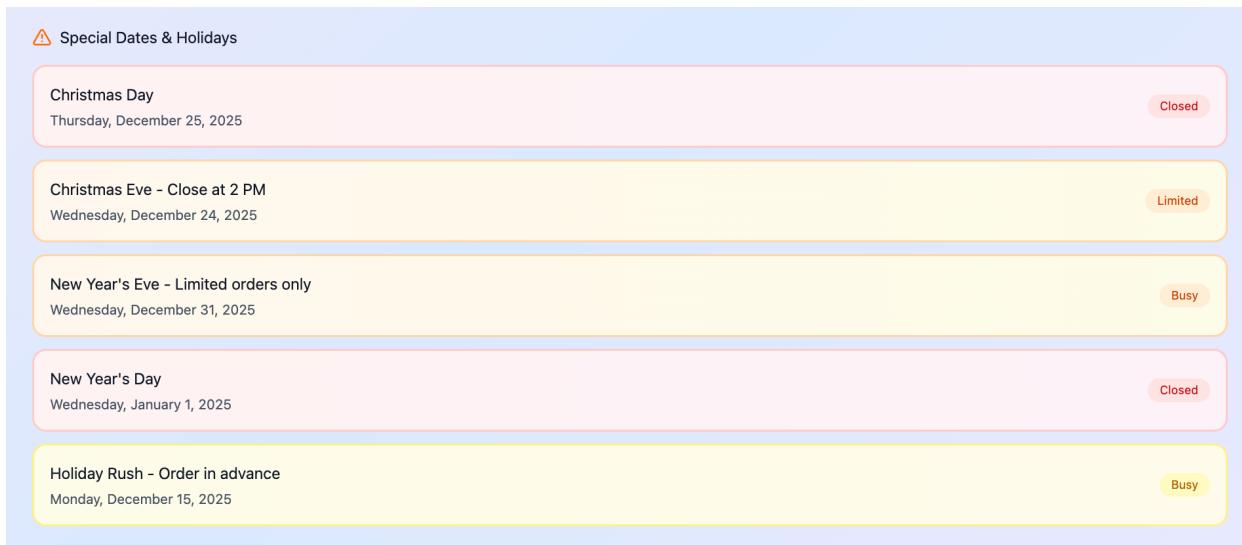
```

- **Result Structure:** Returns the formatted weekly schedule: (day\_of\_week TEXT, time\_range TEXT, status\_tag TEXT, is\_today BOOLEAN).

```
1 ✓ SELECT * FROM fn_get_customer_weekly_schedule( p_shop_id 20);
2
```

Output Result 80

|   | day_of_week | time_range        | status_tag | is_today |
|---|-------------|-------------------|------------|----------|
| 1 | Monday      | 9:00 AM - 5:00 PM | Available  | false    |
| 2 | Tuesday     | 9:00 AM - 6:00 PM | Available  | false    |



## fn\_get\_upcoming\_special\_dates

- Purpose:** Lists upcoming special dates relevant to customers, filtered to future events.
- How It Works (Logic):** Selects special dates where date >= CURRENT\_DATE. Formats the date string (e.g., "Monday, July 04, 2026") and flags the closure status.
- Code:**
- ```
create function fn_get_upcoming_special_dates(p_shop_id integer)
    returns TABLE(note text, display_date text, status text, is_closed boolean)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            sd.note::TEXT,
```

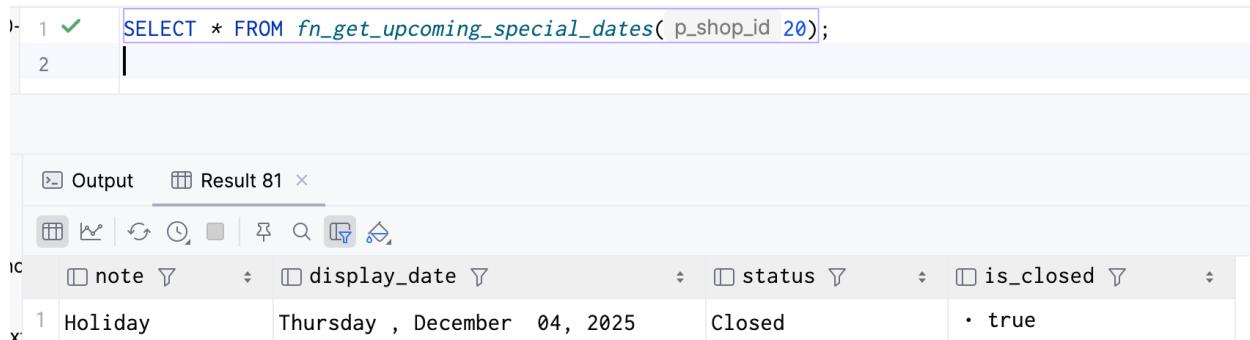
```

        TO_CHAR(sd.date, 'Day, Month DD, YYYY')::TEXT AS display_date,
        sd.status::TEXT,
        (sd.status = 'Closed') AS is_closed
    FROM
        shop_special_date sd
    WHERE
        sd.shop_id = p_shop_id
        AND sd.date >= CURRENT_DATE -- Only show future/today dates
    ORDER BY
        sd.date ASC
    LIMIT 10; -- Show next 10 special events
END;
$$;

alter function fn_get_upcoming_special_dates(integer) owner to root;

```

- **Result Structure:** Returns the next 10 upcoming special dates: (note TEXT, display\_date TEXT, status TEXT, is\_closed BOOLEAN).



The screenshot shows a PostgreSQL terminal window. The command entered is:

```
1 ✓ 1 SELECT * FROM fn_get_upcoming_special_dates( p_shop_id 20);
```

The result set is displayed in a table titled "Result 81". The columns are note, display\_date, status, and is\_closed. There is one row returned:

note	display_date	status	is_closed
Holiday	Thursday , December 04, 2025	Closed	· true

## 5. Finance Model

The proposed monthly subscription fee for the complete platform is **700 - 1,000 Baht**. This fee provides access to all major features: the schedule, detailed order forms, the order tracking dashboard with reminders, and the complete financial tracking system. This pricing is designed to be affordable and provide significant ROI by saving the baker time and reducing errors.

---

## 6. Conclusion and Future Work

The Shop Management Platform addresses critical time-sinks and organizational issues faced by small, custom-order bakeries. By digitizing order intake, inventory tracking, and staff management, the platform directly contributes to increased efficiency and better financial oversight, allowing bakers to focus on their craft.

### 6.1 Project Repository and Assets

- **Git Repository Link:** [Insert your GitHub/GitLab link here]
- **Figma Design Link:** <https://www.figma.com/make/NvltaarrBjOCqKH3QO1Jtq/Login-Page-UI-Design?t=fzYGwgXbulMi0beq-1>