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 Databse 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

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

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	**Mandatory fields** ensure all necessary information (date, size, flavor, reference pictures) is collected in a single submission.

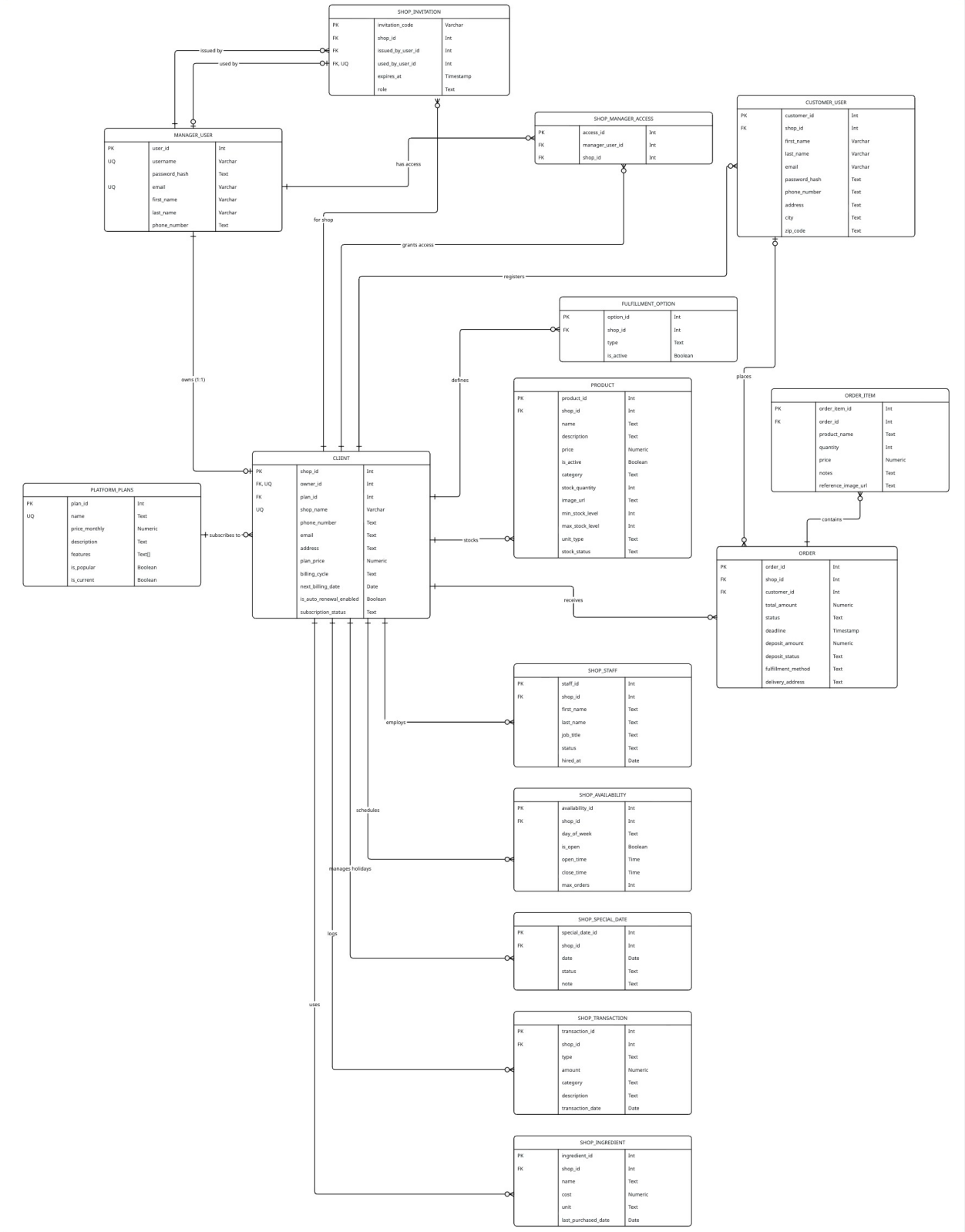
Losing Track of Orders and Deadlines	Order Tracking Dashboard with Reminders	Provides a clear, at-a-glance view of all orders and their status, with automated reminders to prevent missed deadlines.
Hard Time Tracking Money and Costs	Digital Income and Cost Tracker	Enables digital logging of all financial transactions for easy access to **monthly profits** and detailed **ingredient cost tracking**.
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.


SHOP_INVITATION
PK invitation_code Varchar
FK shop_id Int
FK issued_by_user_id Int
FK, UQ used_by_user_id Int
expires_at Timestamp
role Text
issued by
used by
MANAGER_USER
PK user_id Int
UQ username Varchar
password_hash Text
UQ email Varchar
first_name Varchar
last_name Varchar
phone_number Text
has access
SHOP_MANAGER_ACCESS
PK access_id Int
FK manager_user_id Int
FK shop_id Int
CUSTOMER_USER
PK customer_id Int
FK shop_id Int
first_name Varchar
last_name Varchar
email Varchar
password_hash Text
phone_number Text
address Text
city Text
zip_code Text
for shop
grants access
registers
FULFILLMENT_OPTION
PK option_id Int
FK shop_id Int
type Text
is_active Boolean
owns (1:1)
defines
places
PRODUCT
PK product_id Int
FK shop_id Int
name Text
description Text
price Numeric
is_active Boolean
category Text
stock_quantity Int
image_url Text
min_stock_level Int
max_stock_level Int
unit_type Text
stock_status Text
ORDER_ITEM
PK order_item_id Int
FK order_id Int
product_name Text
quantity Int
price Numeric
notes Text
reference_image_url Text
contains
CLIENT
PK shop_id Int
FK, UQ owner_id Int
FK plan_id Int
UQ shop_name Varchar
phone_number Text
email Text
address Text
plan_price Numeric
billing_cycle Text
next_billing_date Date
is_auto_renewal_enabled Boolean
subscription_status Text
PLATFORM_PLANS
PK plan_id Int
UQ name Text
price_monthly Numeric
description Text
features Text[]
is_popular Boolean
is_current Boolean
subscribes to
stocks
receives
ORDER
PK order_id Int
FK shop_id Int
FK customer_id Int
total_amount Numeric
status Text
deadline Timestamp
deposit_amount Numeric
deposit_status Text
fulfillment_method Text
delivery_address Text
SHOP_STAFF
PK staff_id Int
FK shop_id Int
first_name Text
last_name Text
job_title Text
status Text
hired_at Date
employs
SHOP_AVAILABILITY
PK availability_id Int
FK shop_id Int
day_of_week Text
is_open Boolean
open_time Time
close_time Time
max_orders Int
schedules
manages holidays
SHOP_SPECIAL_DATE
PK special_date_id Int
FK shop_id Int
date Date
status Text
note Text
logs
SHOP_TRANSACTION
PK transaction_id Int
FK shop_id Int
type Text
amount Numeric
category Text
description Text
transaction_date Date
uses
SHOP_INGREDIENT
PK ingredient_id Int
FK shop_id Int
name Text
cost Numeric
unit Text
last_purchased_date Date

manager_user

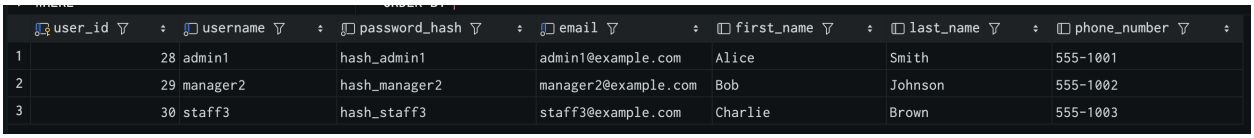
Purpose: Stores all platform-level user accounts (managers, owners, admins). These are the users who manage shops, staff accounts, customers, etc.

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

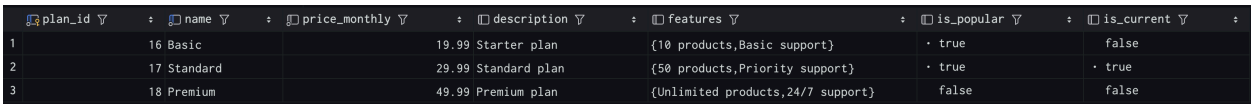
- name: Unique plan name
- price_monthly: Cost of the plan
- features: Array of text listing available features
- is_popular, is_current: Flags for display or filtering purposes

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	20
2	25	29	21
3	26	30	20

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.

shop_invitation

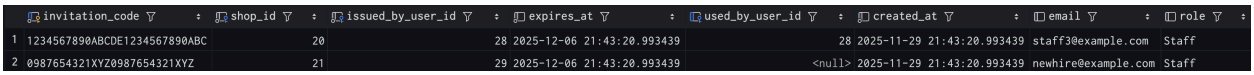
Purpose: Manages invitations to shops (for adding staff/managers).

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

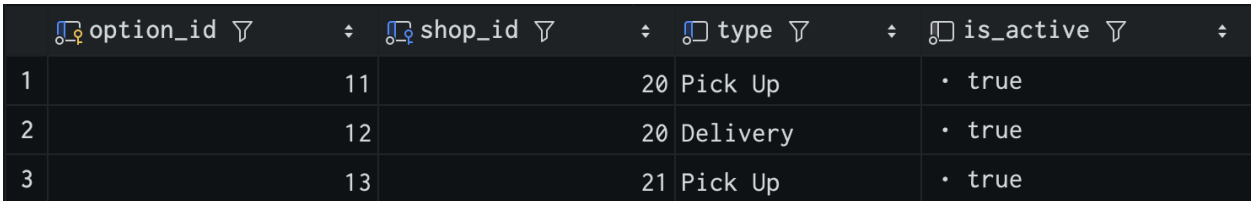
Purpose: Defines available order fulfillment methods per shop (e.g., Pickup, Delivery).

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 +00:0	Home Decor	20
2	17	21	Wireless Mouse	Ergonomic mouse	25.50	true	2025-11-29 21:43:56.052841 +00:0	Electronics	80

image_url	updated_at	min_stock_level	max_stock_level	unit_type	last_restocked_at	stock_status
https://example.com/candle.jp	2025-11-29 21:43:56.052841 +00:0	5	50	pieces	2025-11-26 21:43:56.052841 +00:0	Medium
https://example.com/mouse.jpg	2025-11-29 21:43:56.052841 +00:0	10	100	pieces	2025-11-24 21:43:56.052841 +00:0	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 +00:0	2025-12-01 21:45:39.442277 +00:0
2	41	21	22	25.50	Awaiting Confirmation	2025-11-29 21:45:39.442277 +00:0	2025-11-30 21:45:39.442277 +00:0

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.jp
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	20	Emily	Stone	emily.staff@example.com	555-4001	Cashier	active	2025-11-29
2	13	21	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	20	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	20	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	21	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	20	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	21	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 +00:0	https://example.com/item1.jpg
2	17	41	Wireless Mouse	1	25.50	Test before pickup	2025-11-29 21:47:01.846498 +00:0	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.).

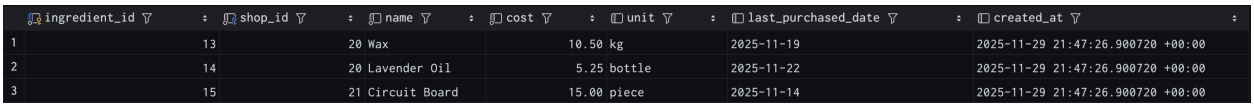
Important Columns

- 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	20	Wax	10.50	kg	2025-11-19	2025-11-29 21:47:26.900720 +00:00
2	14	20	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;
```

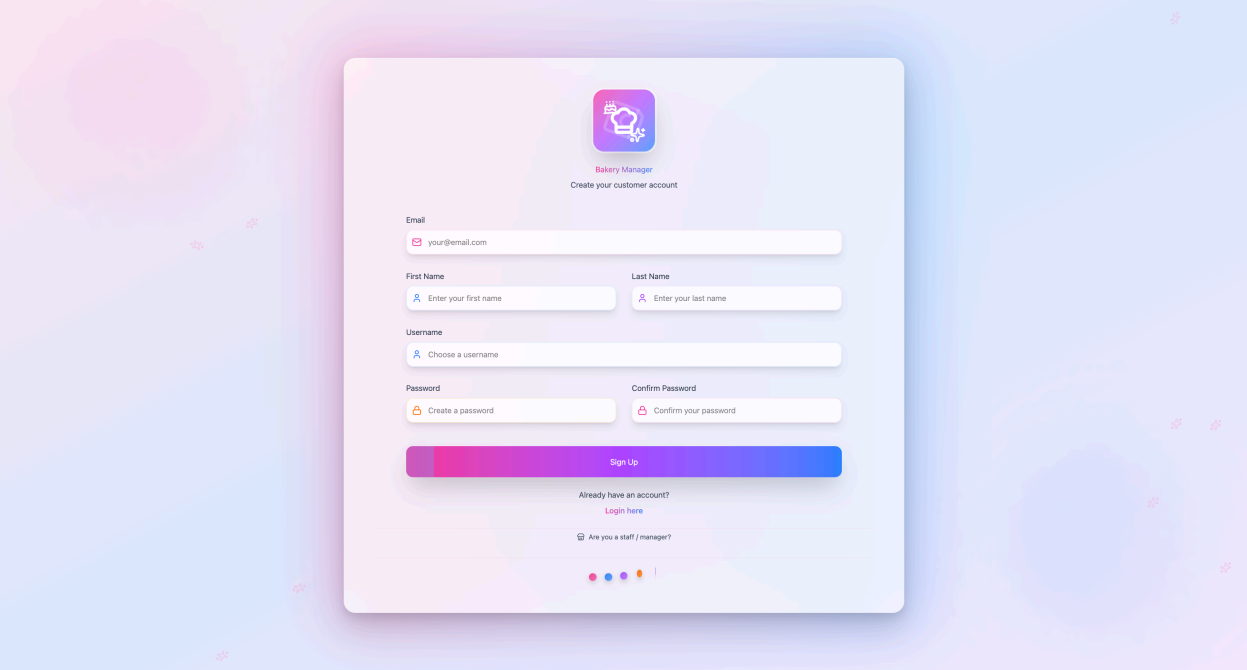
- **Result Structure:** Returns the authorized user's details: 0-(auth_id INTEGER, user_type TEXT, shop_id INTEGER, first_name TEXT).

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

Services > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console_12

Output | Result 67

	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 unique_violation 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. 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 unique_violation 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.

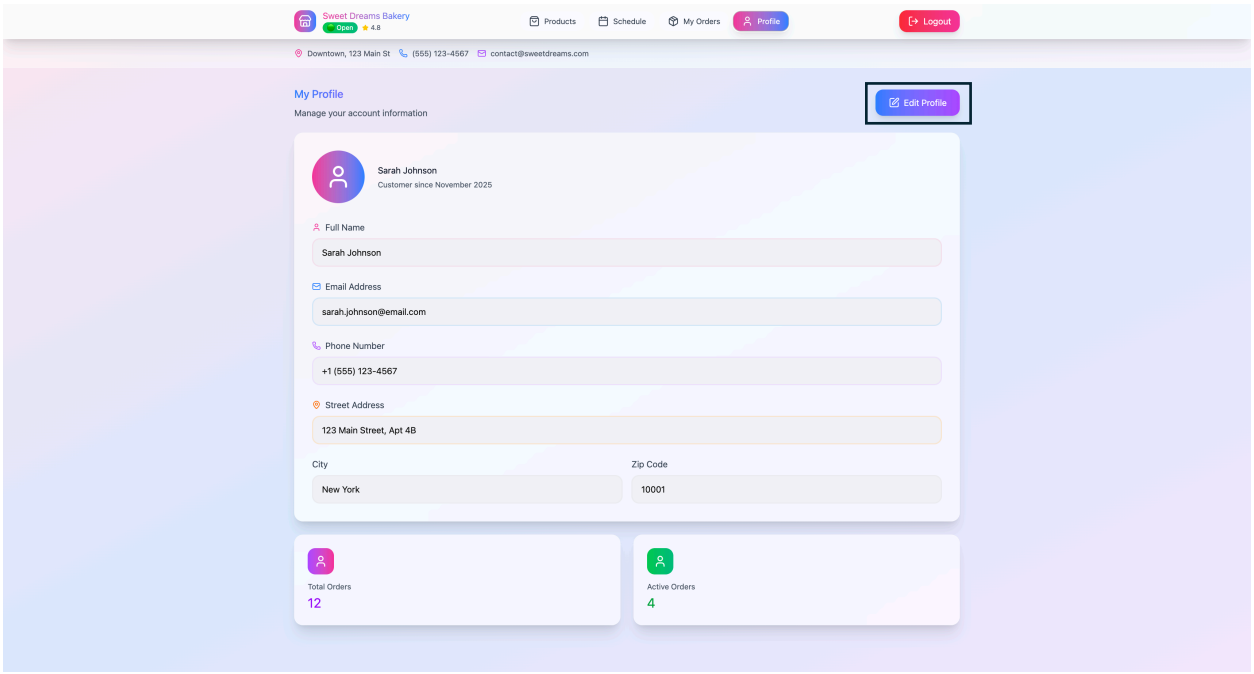
```
SELECT fn_customer_register(
            p_shop_identifier 'Sweet Cake Shop', p_email 'new@customer.com', p_password 'Password1',
            p_first_name 'Jane', p_last_name 'Doe', p_phone_number '0999222333', p_address '123 Street', p_city 'City', p_zip_code '5000'
        );
```

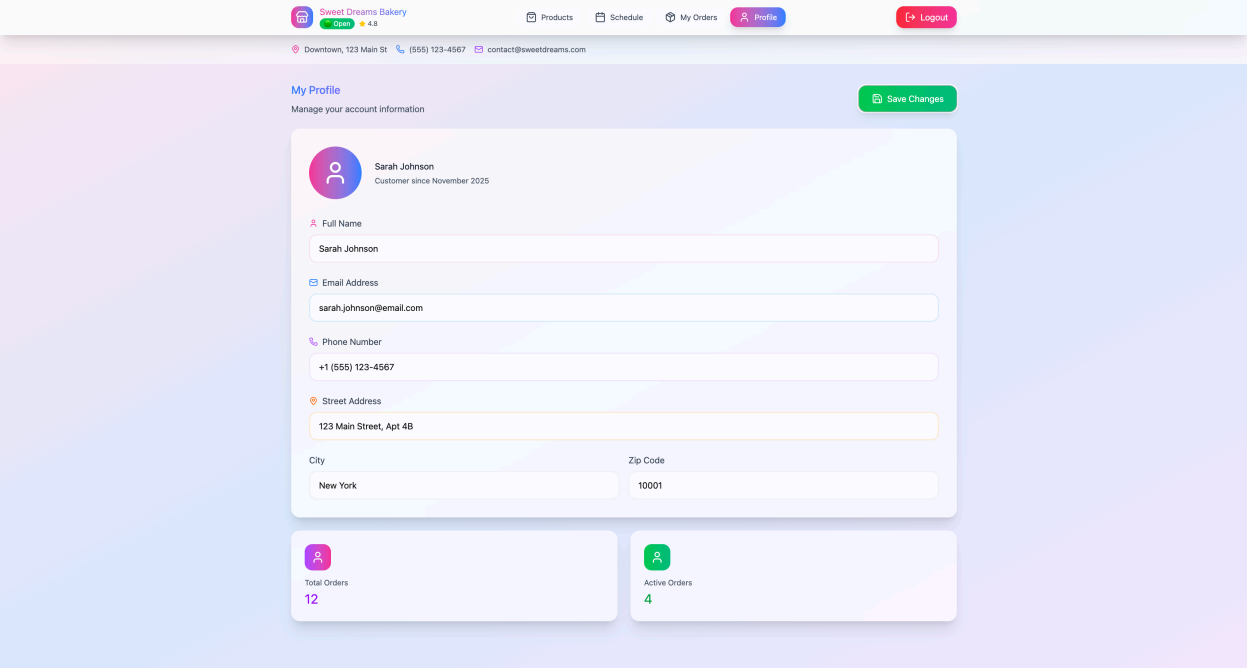
Services > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console_12

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$06$l4P6S.mTuK27WB7Py59OcO8UbwsY./vKENF1kWn0xX3o0zrAmXM

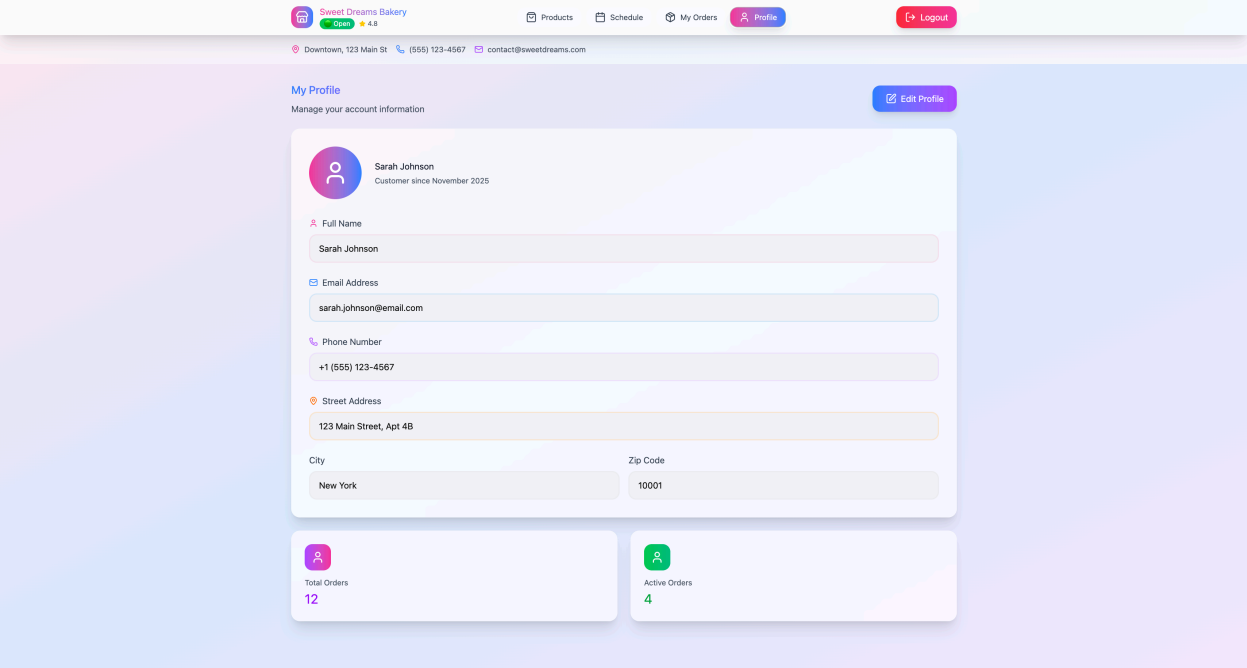




fn_update_customer_profile

- **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:**

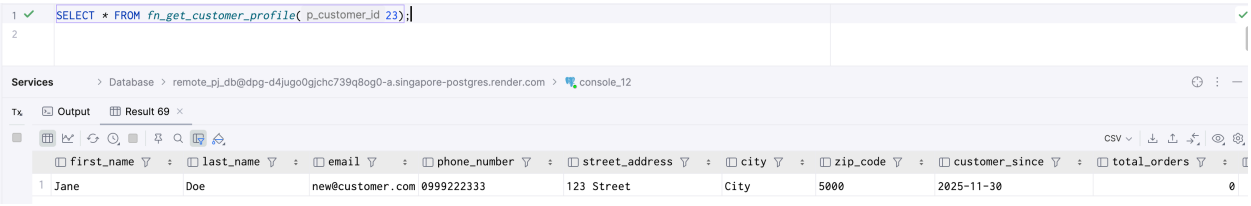
```
create function fn_get_customer_profile(p_customer_id integer)
    returns TABLE(first_name text, last_name text, email text, phone_number text,
street_address text, city text, zip_code text, customer_since date, total_orders
bigint, active_orders bigint)
    language plpgsql
as
$$
BEGIN
    RETURN 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,

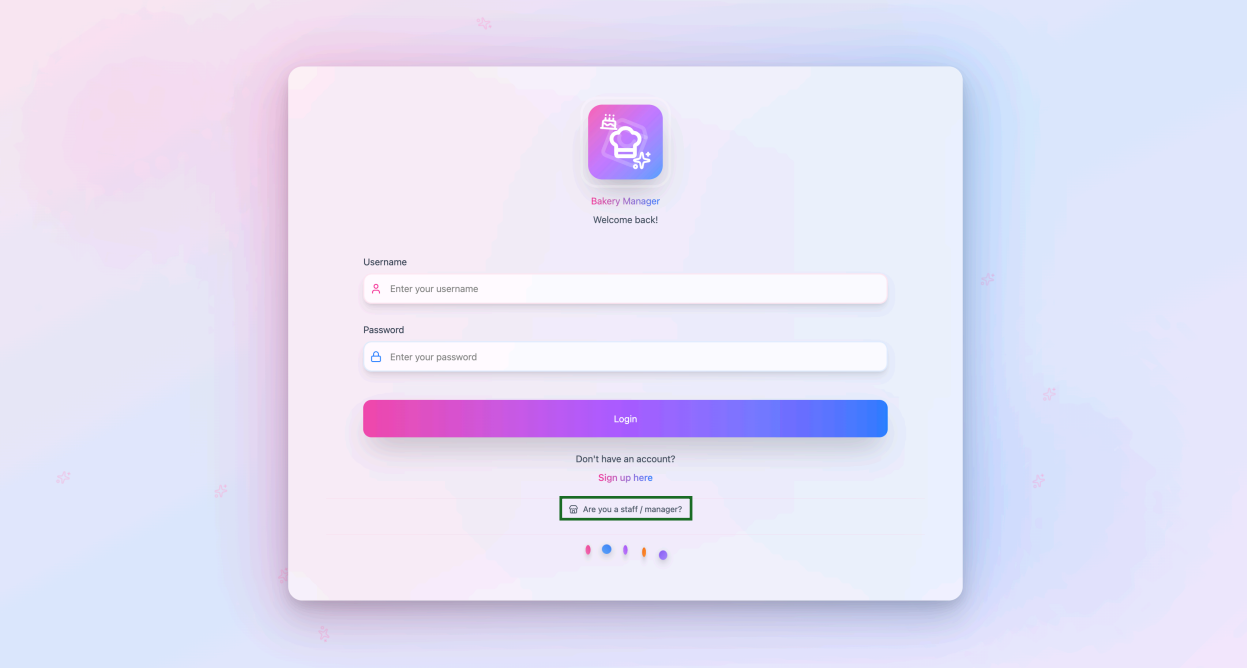
            -- Calculate Active Orders (Not Finished/Cancelled)
            (SELECT COUNT(*) FROM "order" o
             WHERE o.customer_id = cu.customer_id
               AND o.status NOT IN ('Finished', 'Cancelled'))::BIGINT
        FROM
            customer_user cu
        WHERE
            cu.customer_id =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

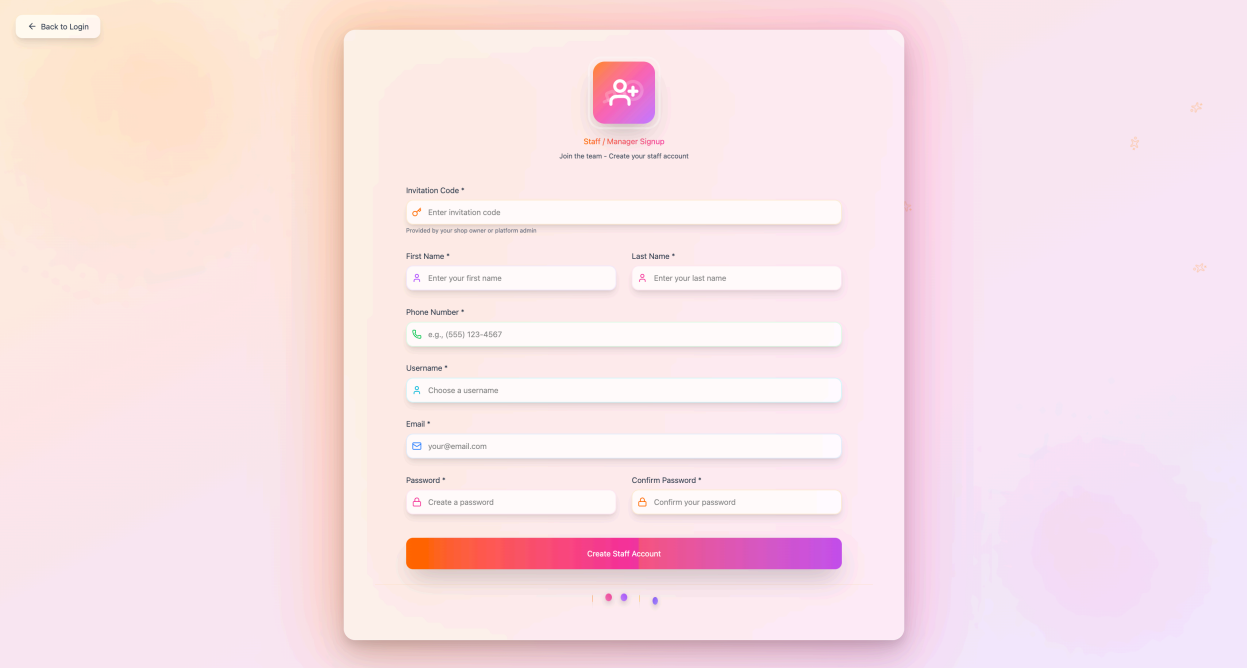
alter function fn_get_customer_profile(integer) owner to root;
```

- **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).



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 unique_violation 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.

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
```

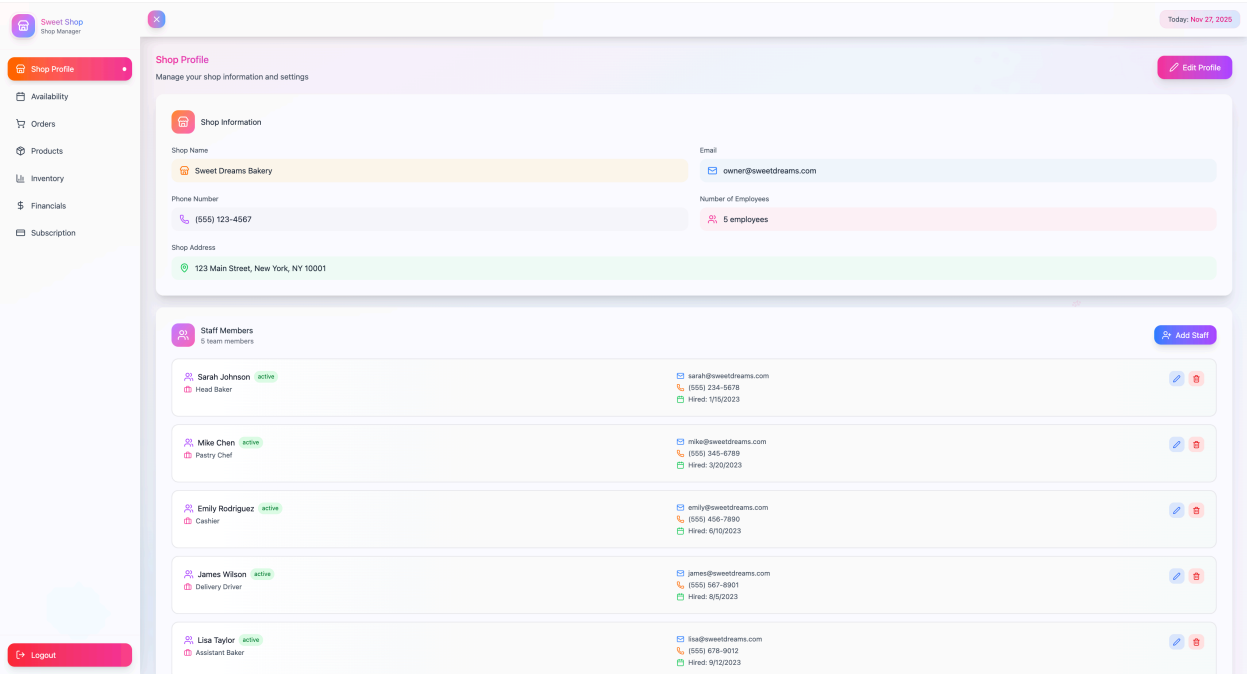
```
            shop_invitation si
                JOIN
            client c ON si.shop_id = c.shop_id
        WHERE
            si.invitation_code = p_invitation_code
          AND si.used_by_user_id IS NULL;

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



fn_get_shop_profile_info

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

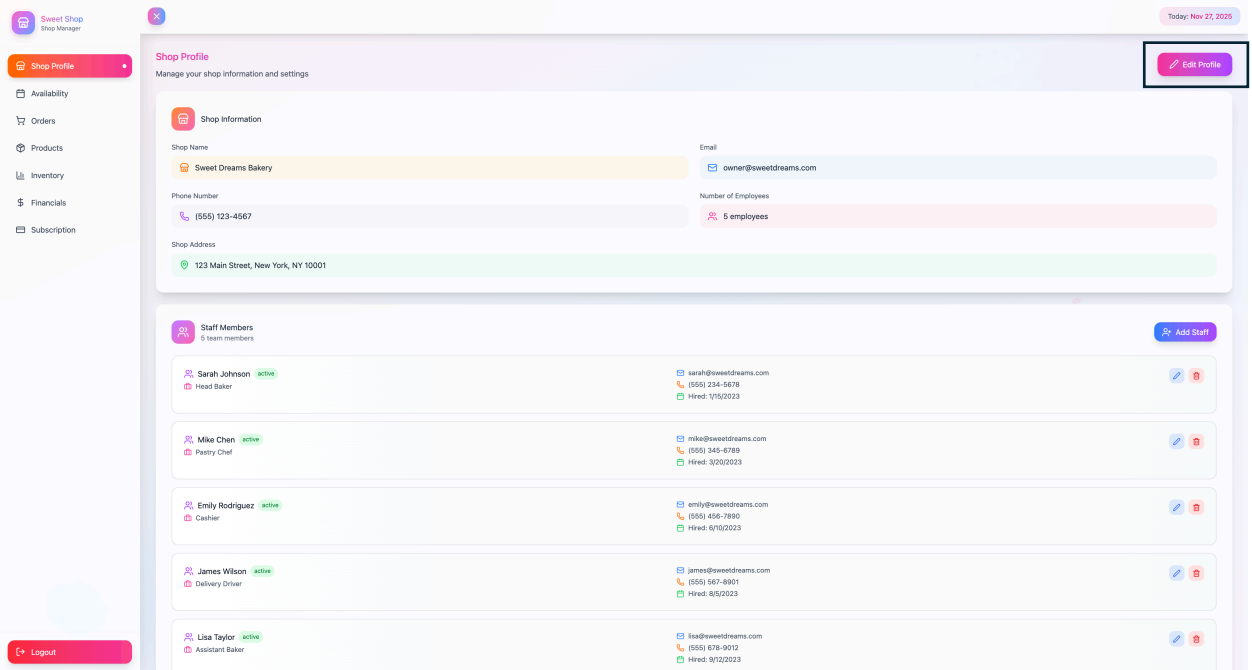
- **How It Works (Logic):** Selects key fields from client. A separate subquery against shop_manager_access counts the distinct manager_user_ids associated with the shop to determine the platform team size (employee_count).
- **Code:**

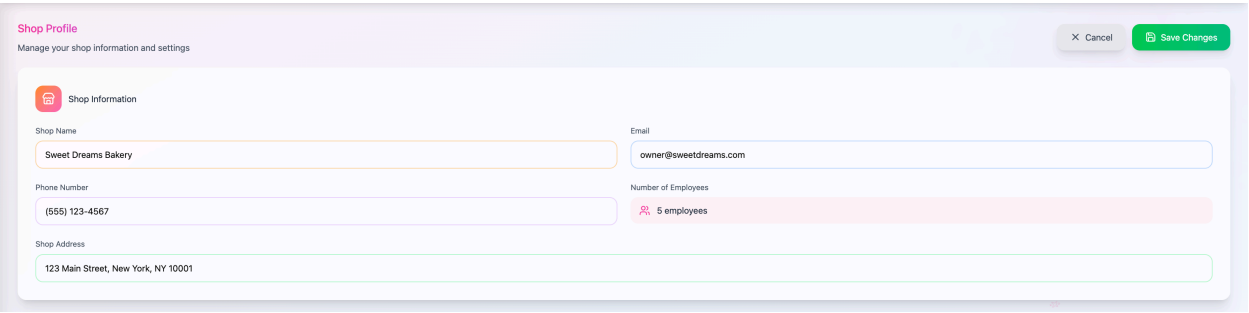
```
create function fn_get_shop_profile_info(p_shop_id integer)
    returns TABLE(shop_name text, email text, phone_number text, shop_address
text, employee_count bigint)
    language plpgsql
as
$$
DECLARE
    v_employee_count BIGINT;
BEGIN
    SELECT COUNT(DISTINCT sma.manager_user_id) INTO v_employee_count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





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).

---

fn_list_shop_managers

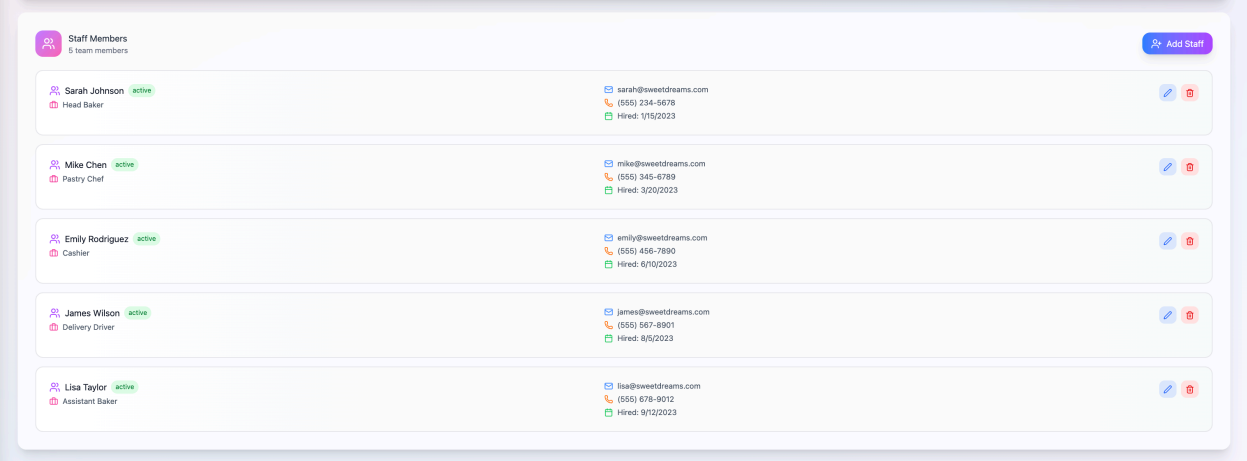
- **Purpose:** Lists all platform managers and staff members who have login access to the specified shop.
- **How It Works (Logic):** Joins manager_user with shop_manager_access using user_id. The query is strictly filtered by shop_id to ensure tenant isolation and only shows authorized personnel. It 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).



fn_get_shop_staff_members

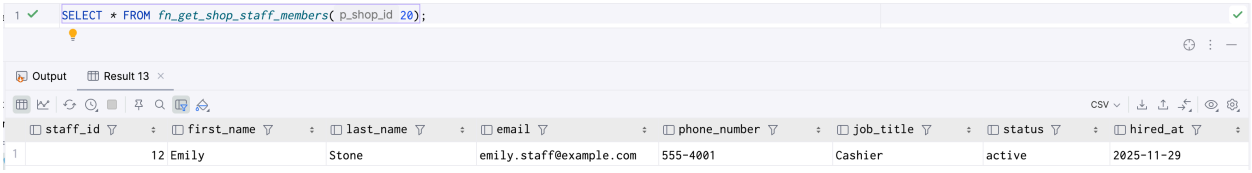
- **Purpose:** Retrieves a list of local (non-platform login) employees of the shop.
- **How It Works (Logic):** Selects all fields from shop_staff, filtering by shop_id, and orders them alphabetically by name.
- **Code:**

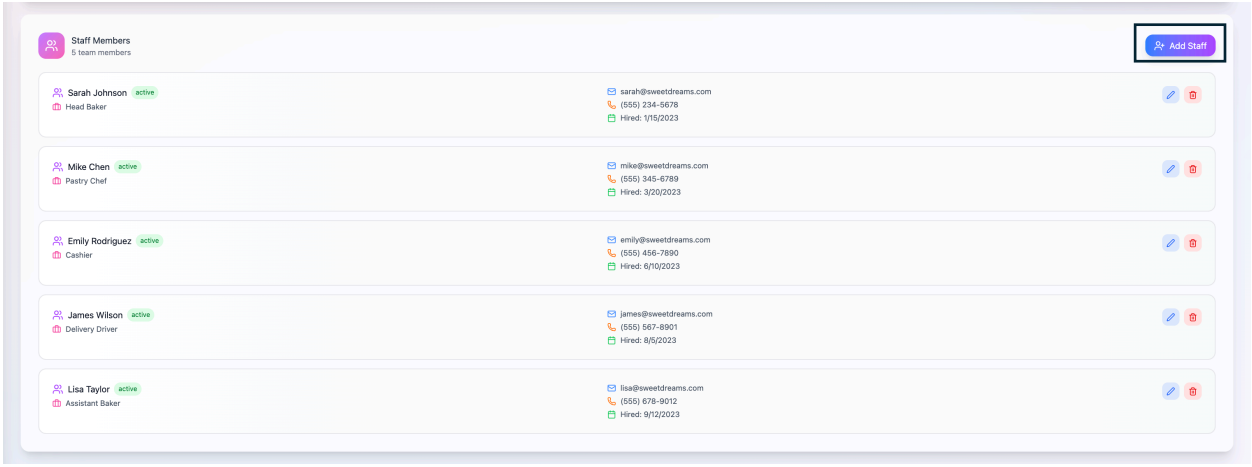
```
create function fn_get_shop_staff_members(p_shop_id integer)
    returns TABLE(staff_id integer, first_name text, last_name text, email text,
phone_number text, job_title text, status text, hired_at date)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            s.staff_id,
  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

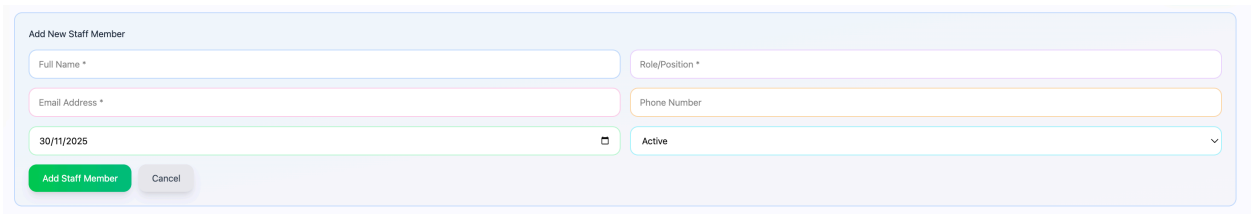
```
        s.status,
        s.hired_at
    FROM
        shop_staff s
    WHERE
        s.shop_id = p_shop_id
    ORDER BY
        s.first_name, s.last_name;
END;
$$;

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).







fn_add_shop_staff_member

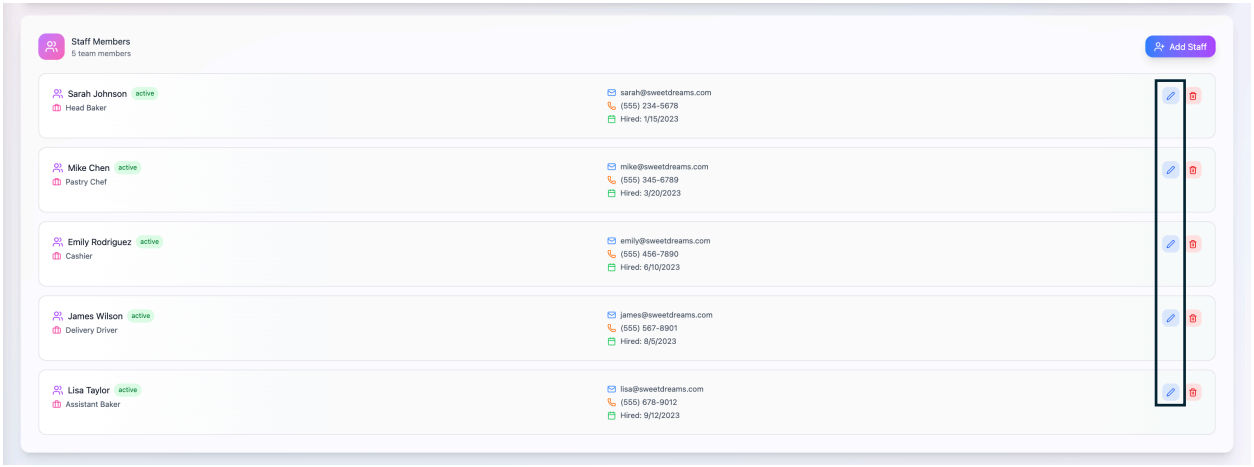
- **Purpose:** Adds a new local employee record to the shop_staff table (for internal tracking/scheduling).
- **How It Works (Logic):** Inserts a new row into shop_staff, setting default values for status ('active') and hired_at (current date if not provided).
- **Code:**

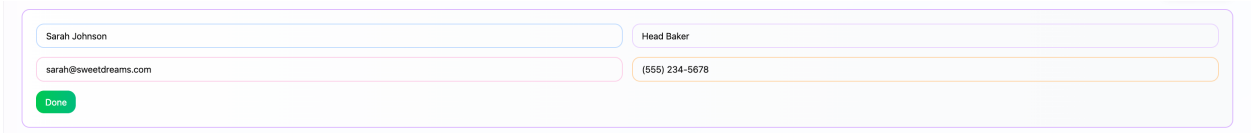
```
create function fn_add_shop_staff_member(p_shop_id integer, p_first_name text,
p_last_name text, p_email text, p_phone_number text, p_job_title text, p_hired_at
date DEFAULT CURRENT_DATE) returns integer
    language plpgsql
as
$$
DECLARE
    v_new_staff_id INT;
BEGIN
    INSERT INTO shop_staff (
        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.





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:**

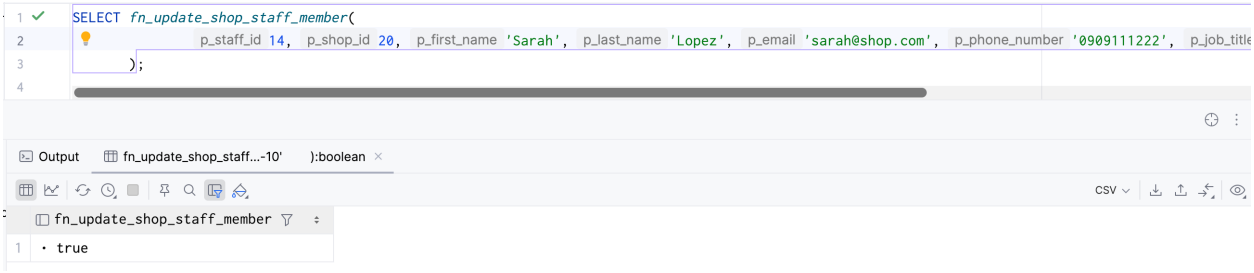
```
create function fn_update_shop_staff_member(p_staff_id integer, p_shop_id integer,
p_first_name text, p_last_name text, p_email text, p_phone_number text,
p_job_title text, p_status text, p_hired_at date) returns boolean
    language plpgsql
as
$$
DECLARE
BEGIN
    UPDATE shop_staff
    SET
    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

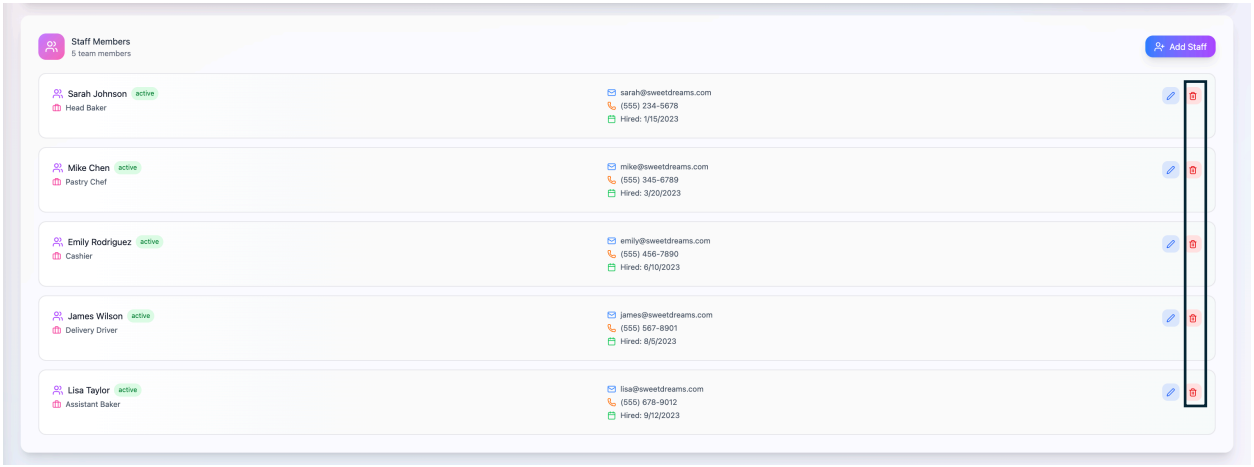
```
        last_name = p_last_name,
        email = p_email,
        phone_number = p_phone_number,
        job_title = p_job_title,
        status = p_status,
        hired_at = p_hired_at
    WHERE
        staff_id = p_staff_id
      AND shop_id = p_shop_id;

    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, text, date) owner to root;
```

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





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:**

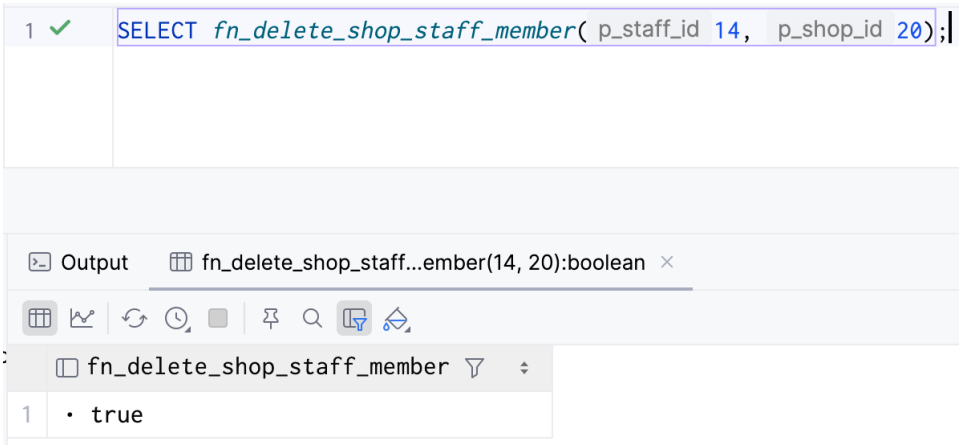
```
create function fn_delete_shop_staff_member(p_staff_id integer, p_shop_id integer)
returns boolean
    language plpgsql
as
$$
DECLARE
    v_deleted_rows INT;
BEGIN
    DELETE FROM shop_staff
    WHERE
        staff_id = p_staff_id
      AND shop_id = p_shop_id;  -- Security check

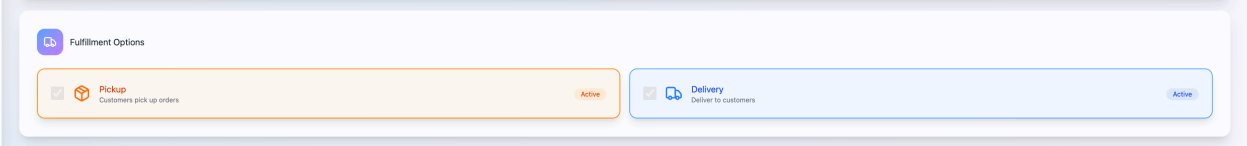
    GET DIAGNOSTICS v_deleted_rows = ROW_COUNT;

    RETURN FOUND;
END;
$$;

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

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





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
$$
DECLARE
BEGIN
    RETURN QUERY
        SELECT
            fo.type AS option_type,
            fo.is_active
        FROM
            fulfillment_option fo
        WHERE
            fo.shop_id = p_shop_id
        ORDER BY
            fo.type;
END;
$$;

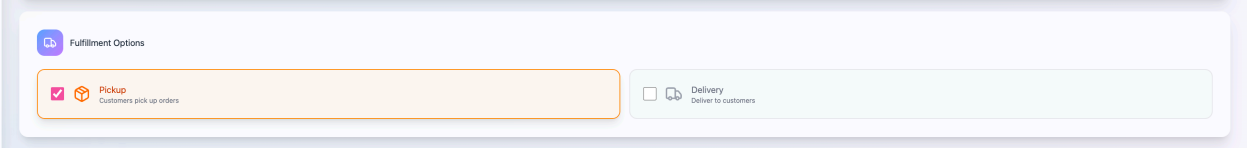
alter function fn_get_fulfillment_options(integer) owner to root;
```

- **Result Structure:** Returns the list of available methods: (option_type TEXT, is_active BOOLEAN).

```
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;

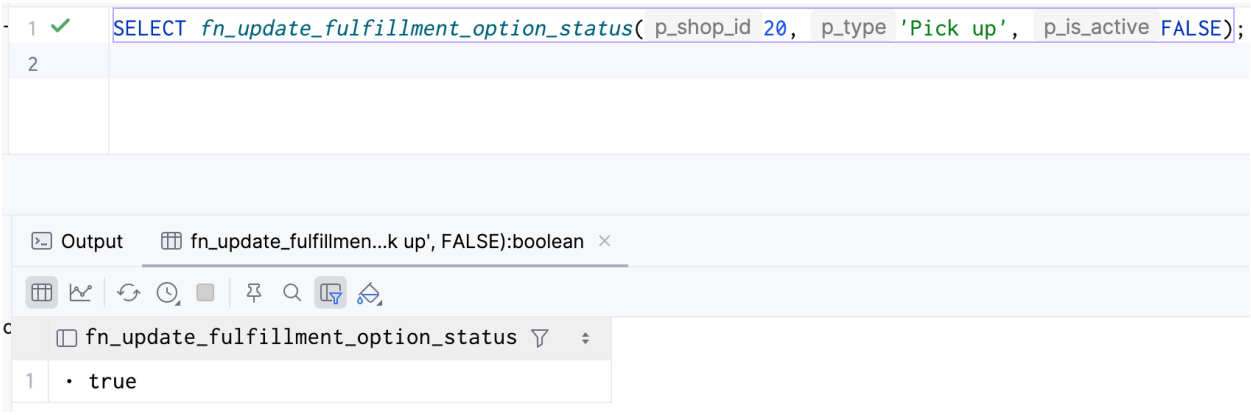
      IF NOT FOUND THEN
          INSERT INTO fulfillment_option (shop_id, type, is_active)
          VALUES (p_shop_id, p_type, p_is_active);
  ```
  ```

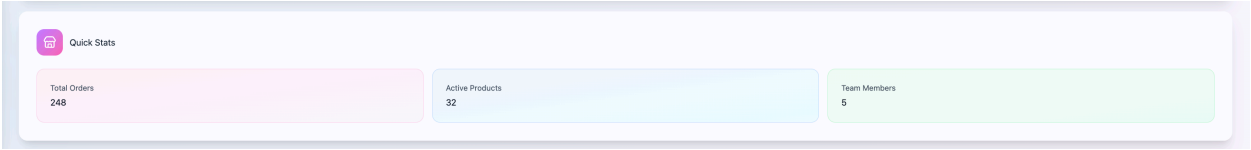
```
    END IF;

    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).





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
```

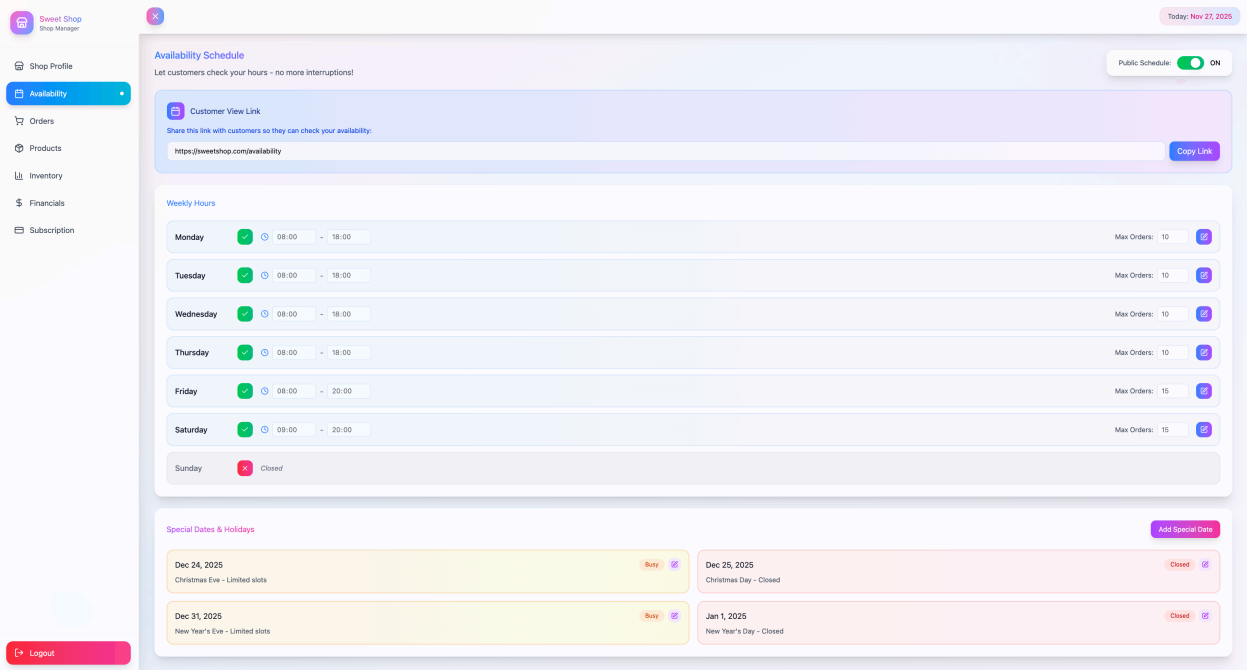
- **Result Structure:** Returns three aggregate counts: (total_orders BIGINT, active_products BIGINT, team_members BIGINT).

```
SELECT * FROM fn_get_shop_quick_stats( p_shop_id 20);
```

Output | Result 20

	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;
END;
$$;

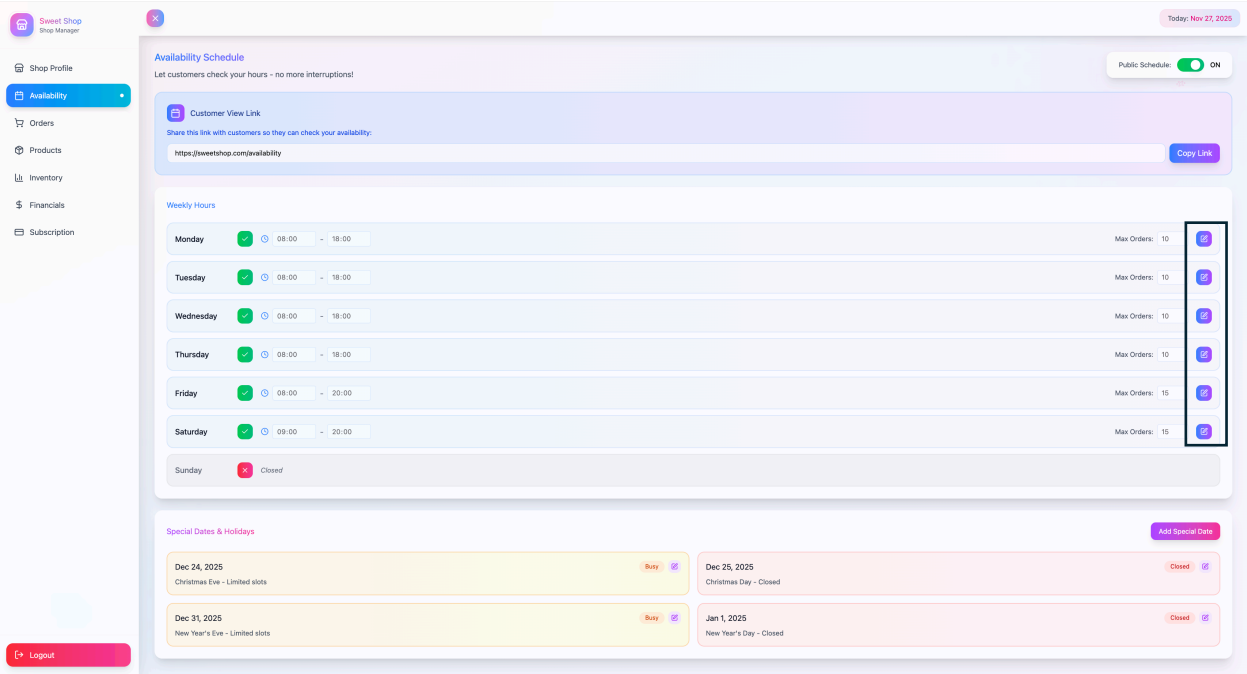
alter function fn_get_shop_availability(integer) owner to root;
```

- **Result Structure:** Returns the weekly schedule: (day_of_week TEXT, is_open BOOLEAN, open_time TIME, close_time TIME, max_orders INTEGER).

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

Output | Result 21

	day_of_week	is_open	open_time	close_time	max_orders
1	Monday	true	09:00:00	18:00:00	20
2	Tuesday	true	09:00:00	18:00:00	20



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:**

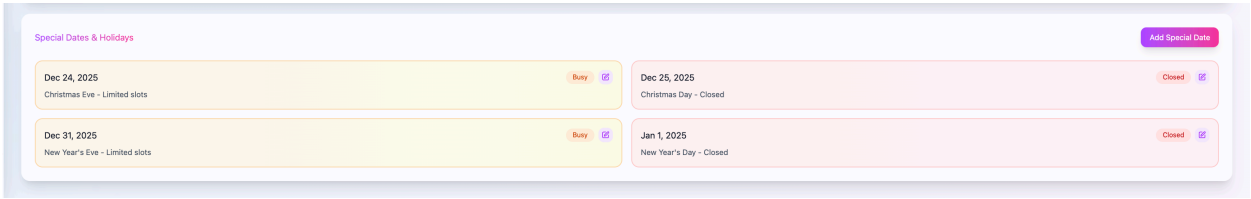
```
create function fn_update_day_availability(p_shop_id integer, p_day_of_week text,
p_is_open boolean, p_open_time time without time zone, p_close_time time without
time zone, p_max_orders integer) returns boolean
    language plpgsql
as
$$
BEGIN
    UPDATE shop_availability
    SET
        is_open = 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.



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
$$
BEGIN
    RETURN QUERY
        SELECT
            sd.special_date_id,
            sd.date,
            sd.status,
            sd.note
        FROM
            shop_special_date sd
        WHERE
            sd.shop_id = p_shop_id
        ORDER BY
            sd.date ASC;
END;
$$;
```

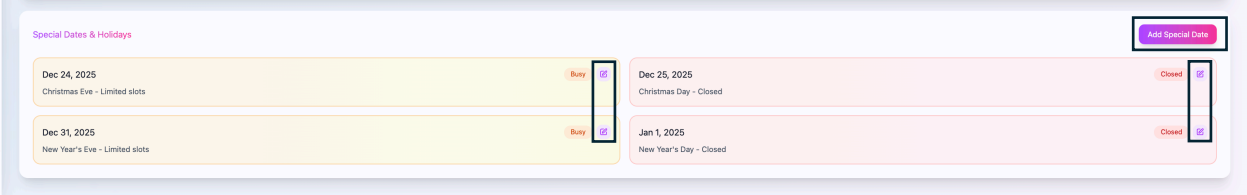
```
alter function fn_get_shop_special_dates(integer) owner to root;
```

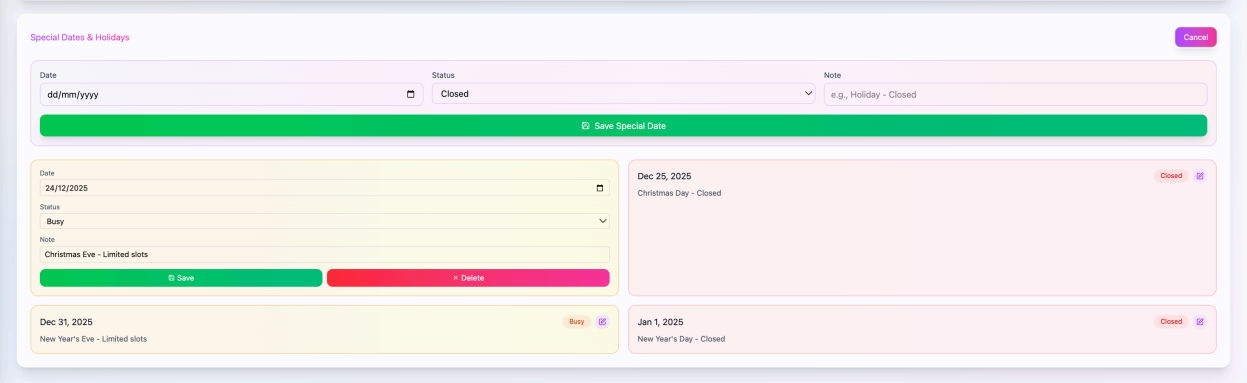
- **Result Structure:** Returns the special date list: (special_date_id INTEGER, date DATE, status TEXT, note TEXT).

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

Output | Result 23

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





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:**

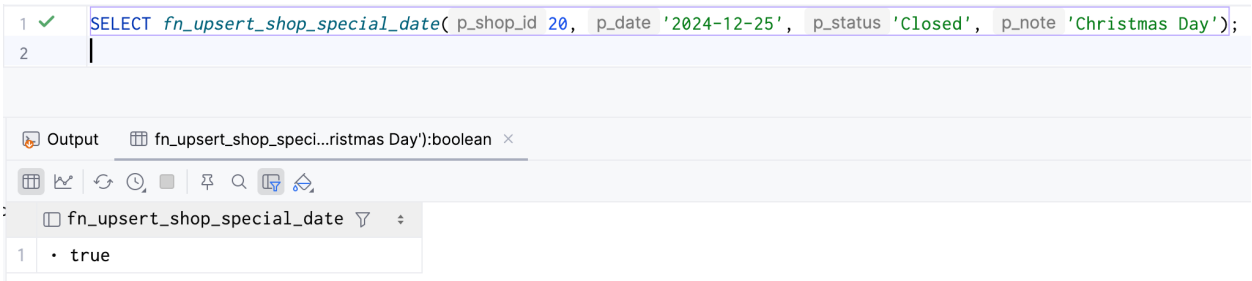
```
create function fn_upsert_shop_special_date(p_shop_id integer, p_date date,
p_status text, p_note text) returns boolean
    language plpgsql
as
$$
BEGIN
    UPDATE shop_special_date
    SET
        status = p_status,
        note = p_note,
        updated_at = NOW()
    WHERE
        shop_id = p_shop_id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

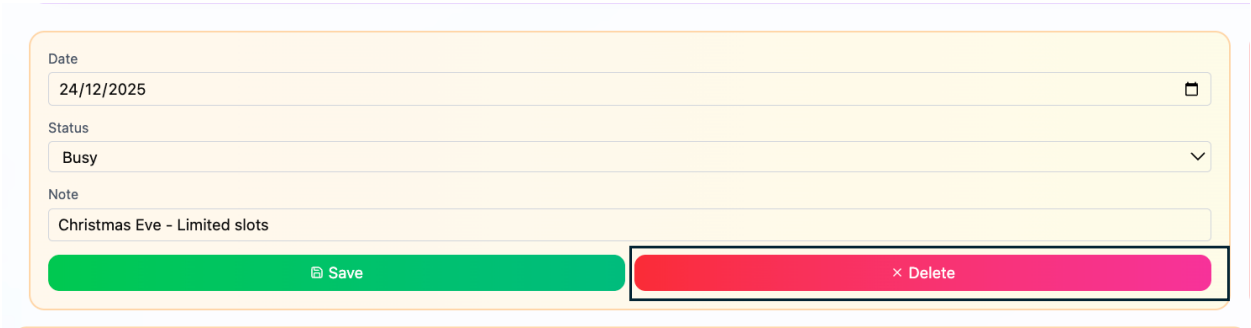
    IF NOT FOUND THEN
        INSERT INTO shop_special_date (shop_id, date, status, note)
        VALUES (p_shop_id, p_date, p_status, p_note);
    END IF;

    RETURN TRUE;
END;
$$;

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

- **Result Structure:** Returns BOOLEAN.





fn_delete_shop_special_date

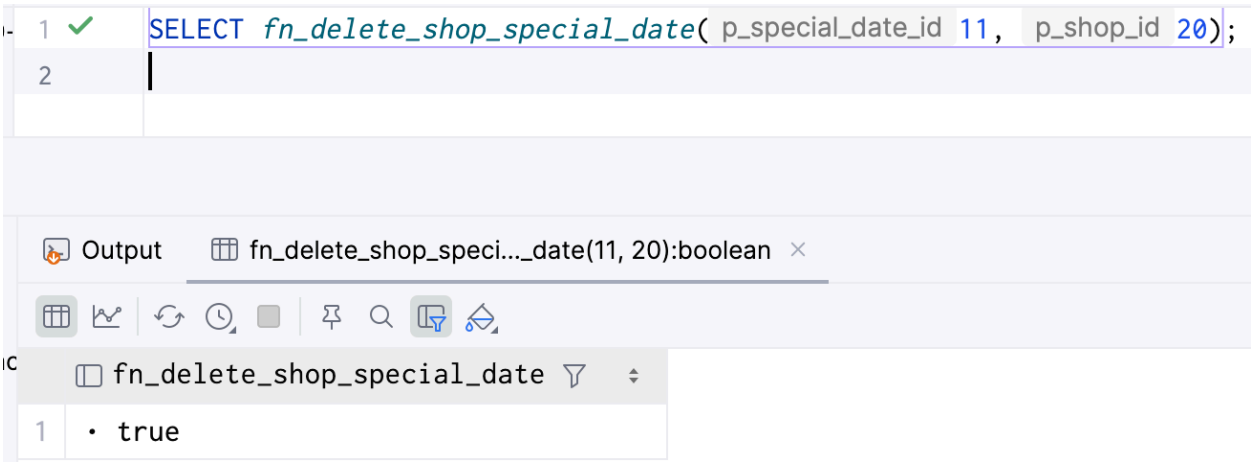
- **Purpose:** Removes a special date entry.
- **How It Works (Logic):** Executes a standard DELETE query, requiring both special_date_id and shop_id for security.
- **Code:**

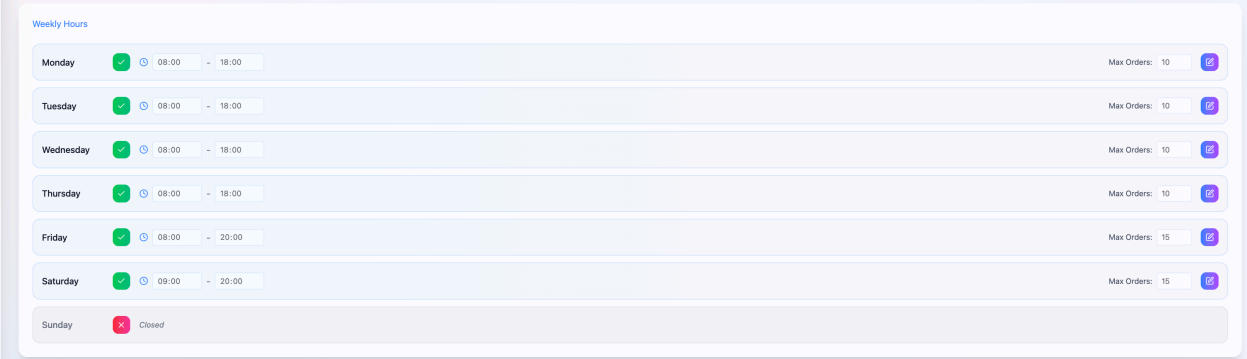
```
create function fn_delete_shop_special_date(p_special_date_id integer, p_shop_id
integer) returns boolean
    language plpgsql
as
$$
BEGIN
    DELETE FROM shop_special_date
    WHERE
        special_date_id = p_special_date_id
      AND shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

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

- **Result Structure:** Returns BOOLEAN.





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
```

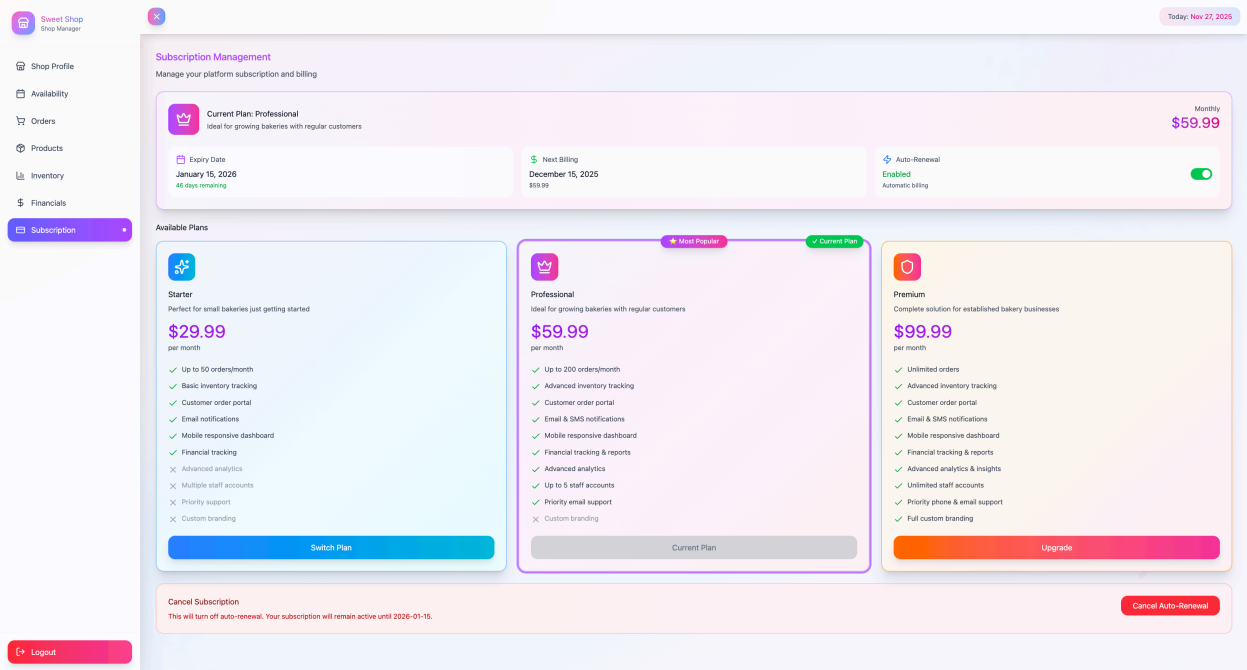
```
            sa.shop_id = p_shop_id
        AND sa.day_of_week = v_today_name;
END;
$$;

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)



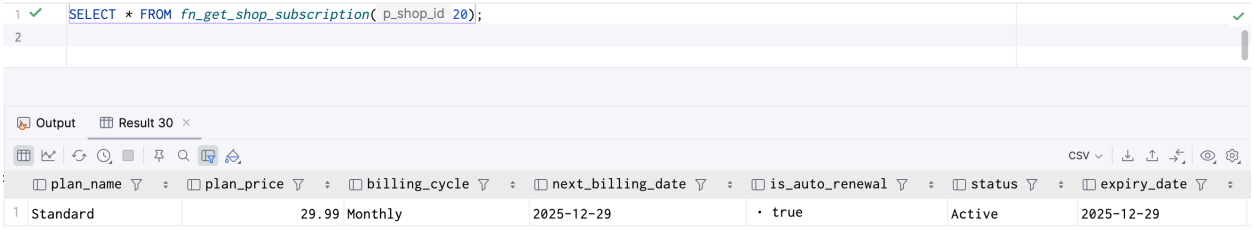
fn_get_shop_subscription

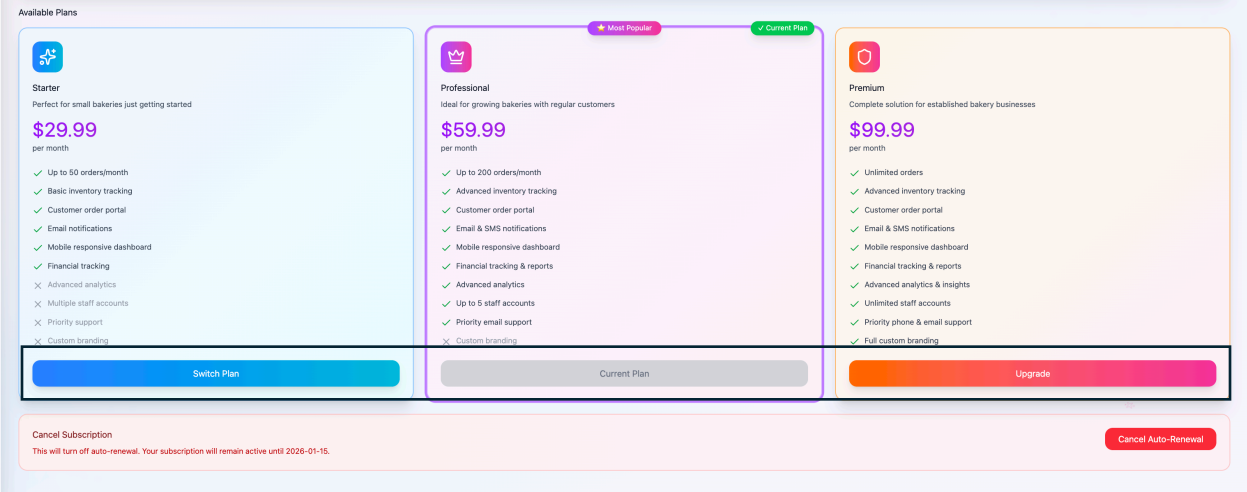
- **Purpose:** Retrieves the client's current subscription details, combining static plan data with the shop's dynamic billing status.
- **How It Works (Logic):** Joins client (for snapshot price, cycle, and status) with platform_plans (for the current plan name). Calculates days_remaining until the 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).





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;

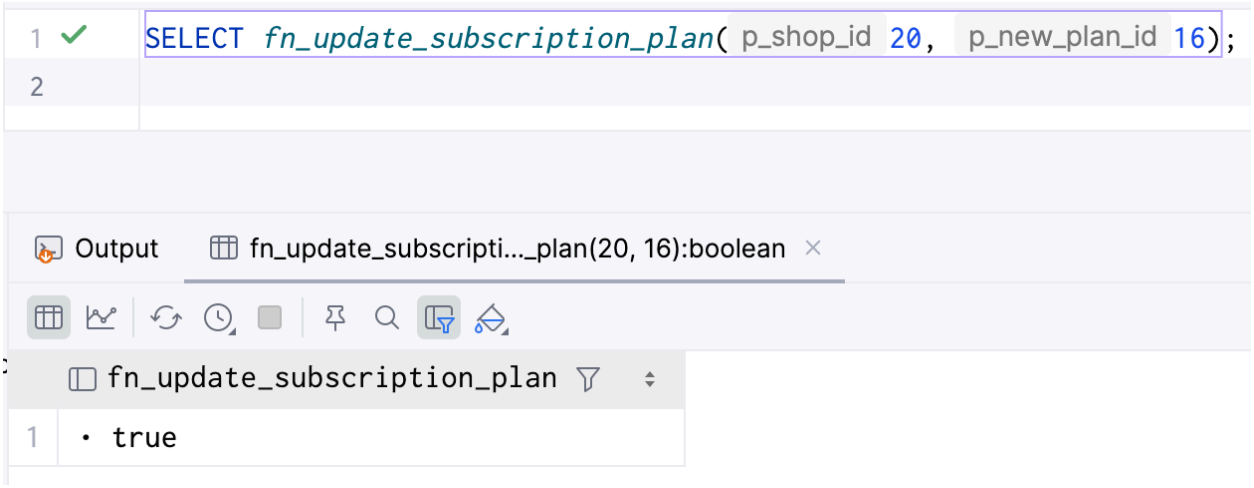
    -- Update client with new plan ID and the NEW price
    UPDATE client
    SET
        plan_id = p_new_plan_id,
        plan_price = v_new_price
    WHERE
```

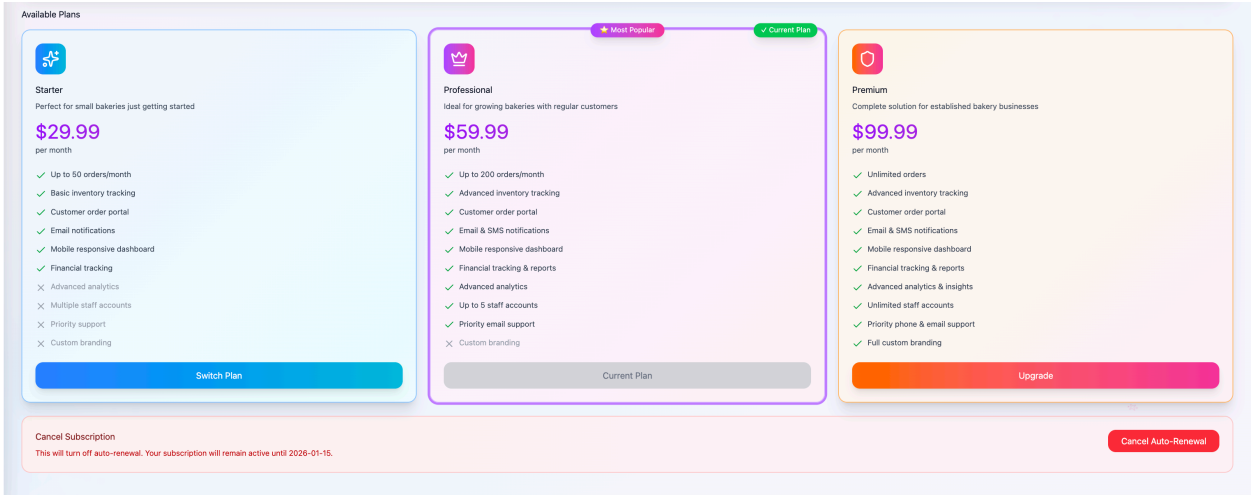
```
        shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

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

- **Result Structure:** Returns BOOLEAN.





fn_get_available_plans

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

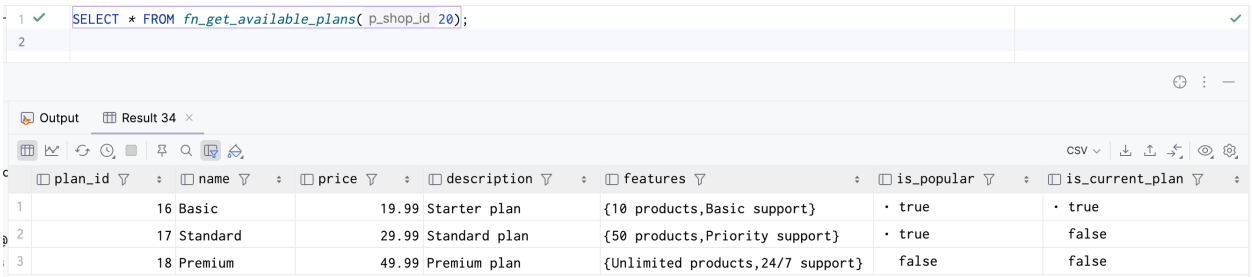
- **How It Works (Logic):** Queries platform_plans. It uses a comparison against the shop's current plan_id to set the is_current_plan flag.
- **Code:**

```
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;

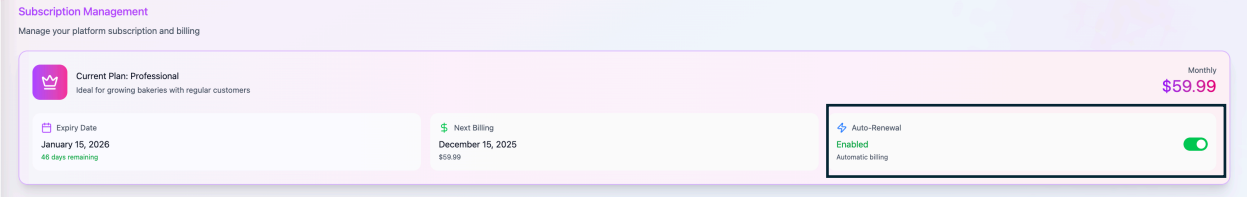
    RETURN QUERY
        SELECT
            p.plan_id,
            p.name::TEXT,
            p.price_monthly AS price,      -- Alias to match return table 'price'
            p.description::TEXT,
            p.features,
            p.is_popular,
            -- Safe boolean comparison
            COALESCE(p.plan_id = 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).



	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;
```

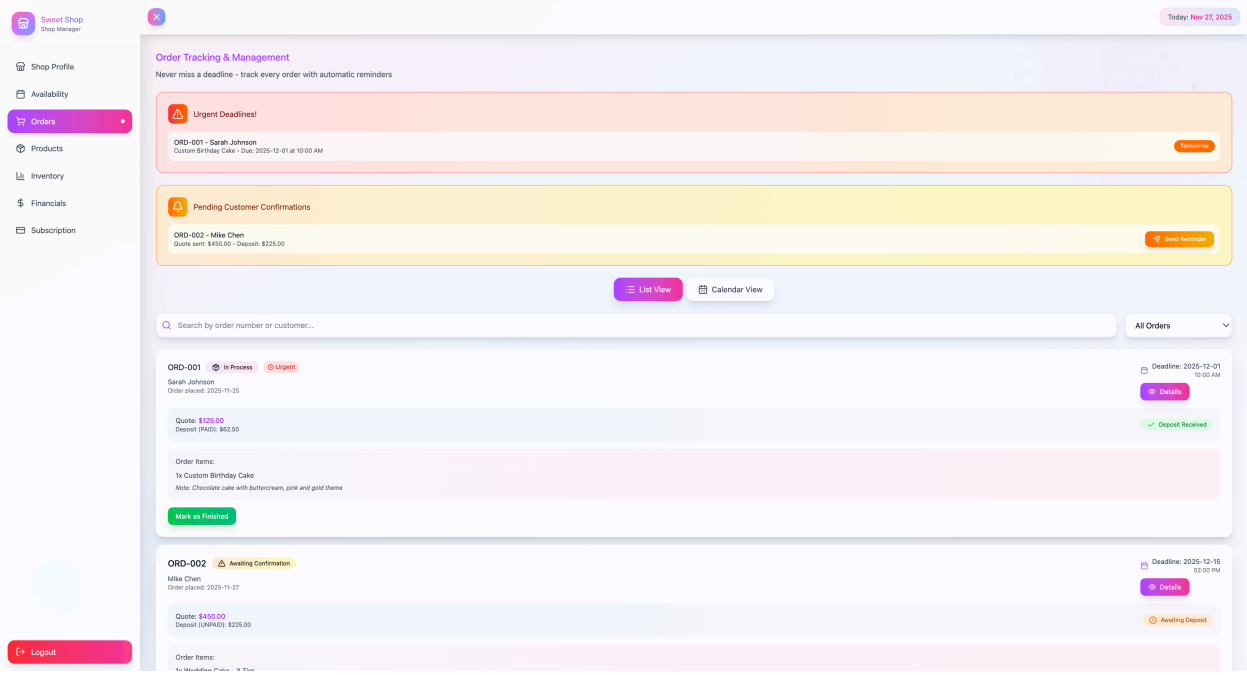
- **Result Structure:** Returns BOOLEAN.

```
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



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.

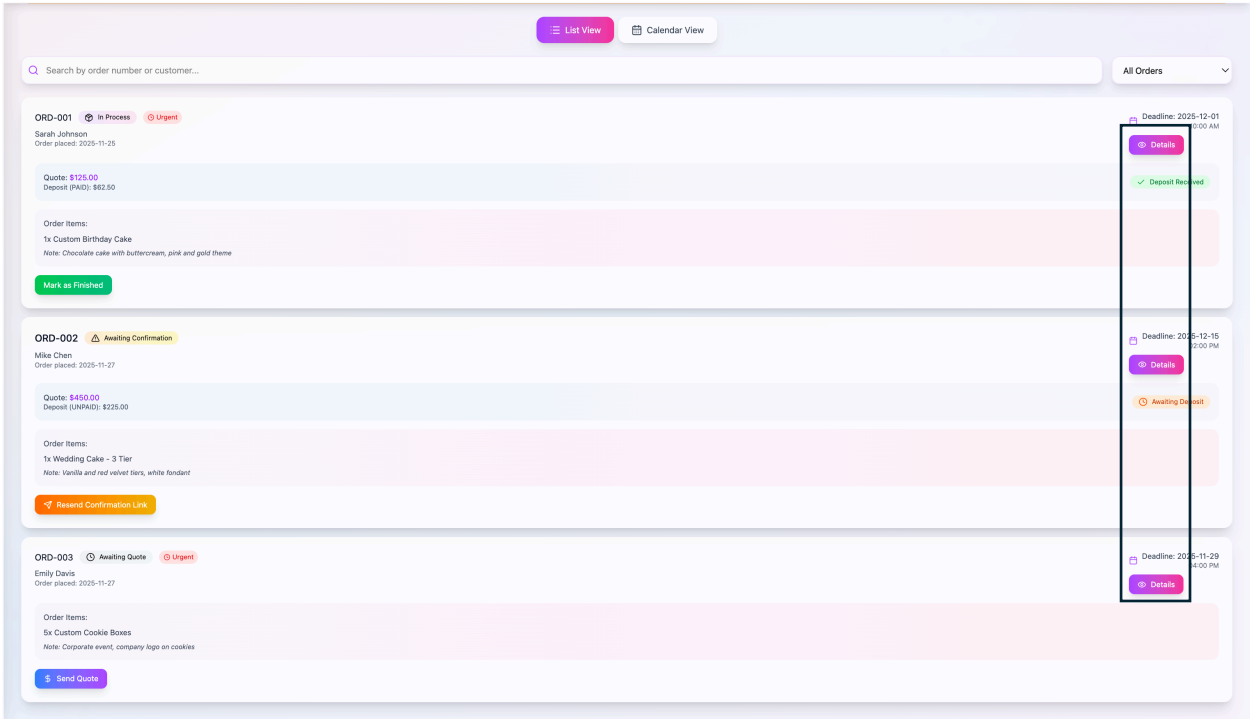
- **Code:**

```
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).



fn_get_order_details

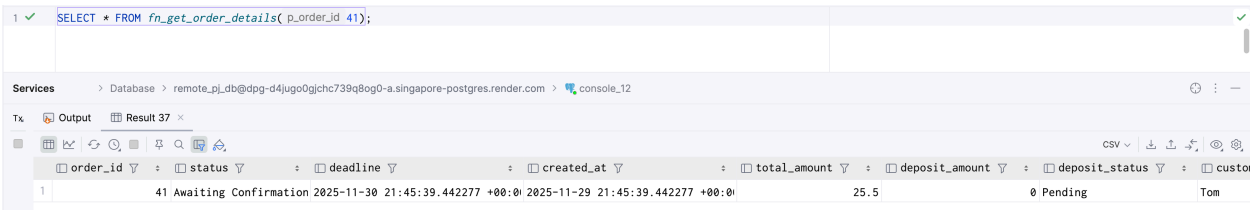
- **Purpose:** Fetches the primary order and customer contact details necessary for order fulfillment.
- **How It Works (Logic):** Joins order with customer_user (LEFT JOIN to support guest orders).

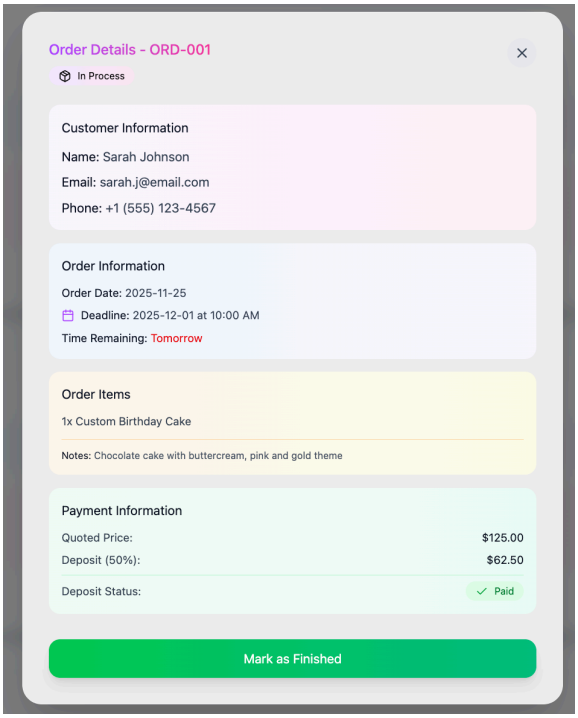
- **Code:**

```
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).





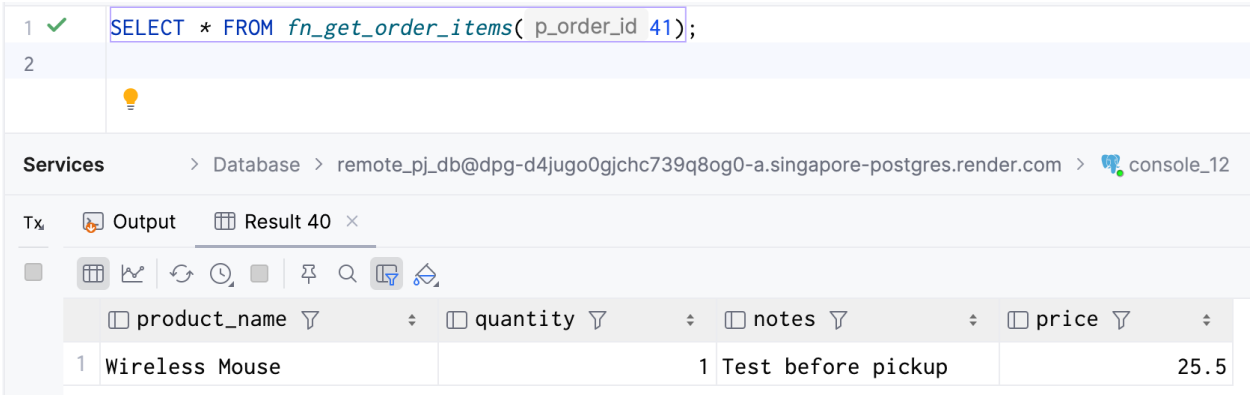
fn_get_order_items

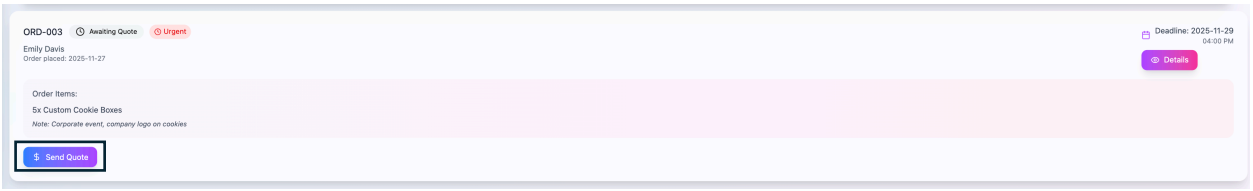
- **Purpose:** Retrieves the list of line items (products, quantities, notes) associated with a specific order.]
- **How It Works (Logic):** Simple query against order_item filtered by order_id.
- **Code:**

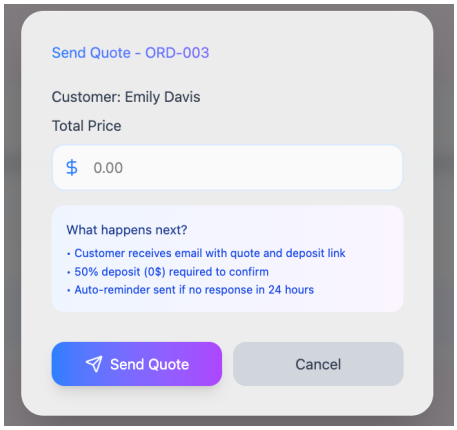
```
create function fn_get_order_items(p_order_id integer)
    returns TABLE(product_name text, quantity integer, notes text, price numeric)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            oi.product_name,
            oi.quantity,
            oi.notes,
            oi.price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

```
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).







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'.

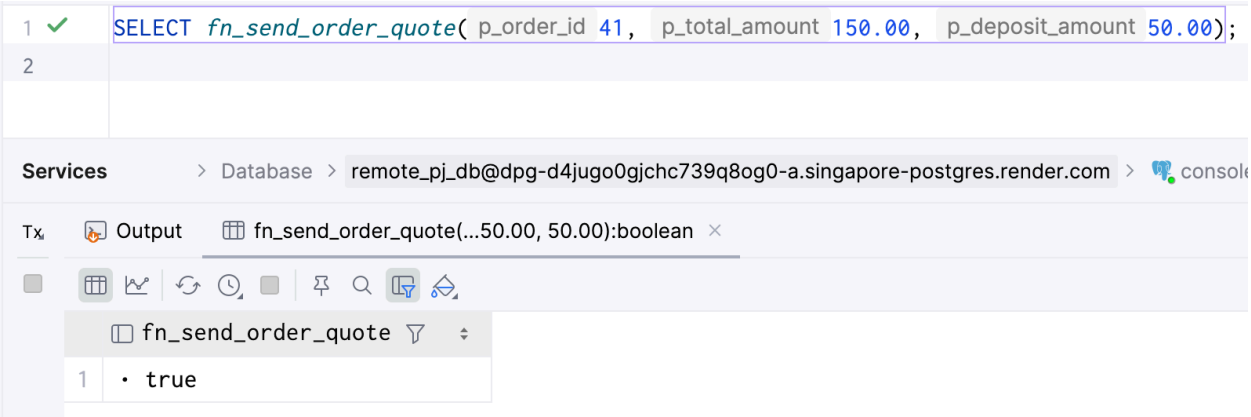
- **Code:**

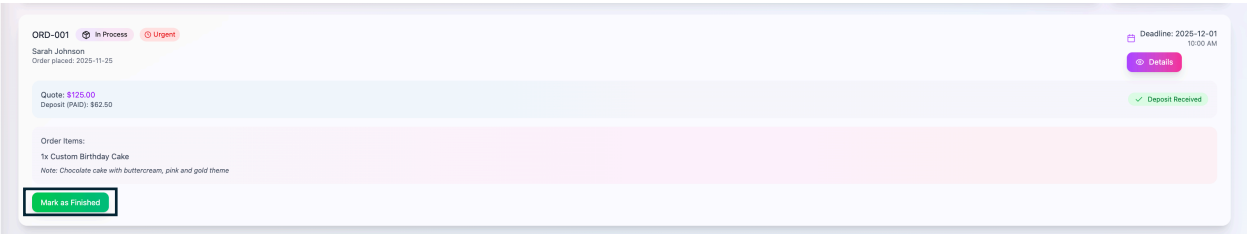
```
create function fn_send_order_quote(p_order_id integer, p_total_amount numeric,
p_deposit_amount numeric) returns boolean
    language plpgsql
as
$$
BEGIN
    UPDATE "order"
    SET
        total_amount = p_total_amount,
        deposit_amount = p_deposit_amount,
        status = 'Awaiting Confirmation',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



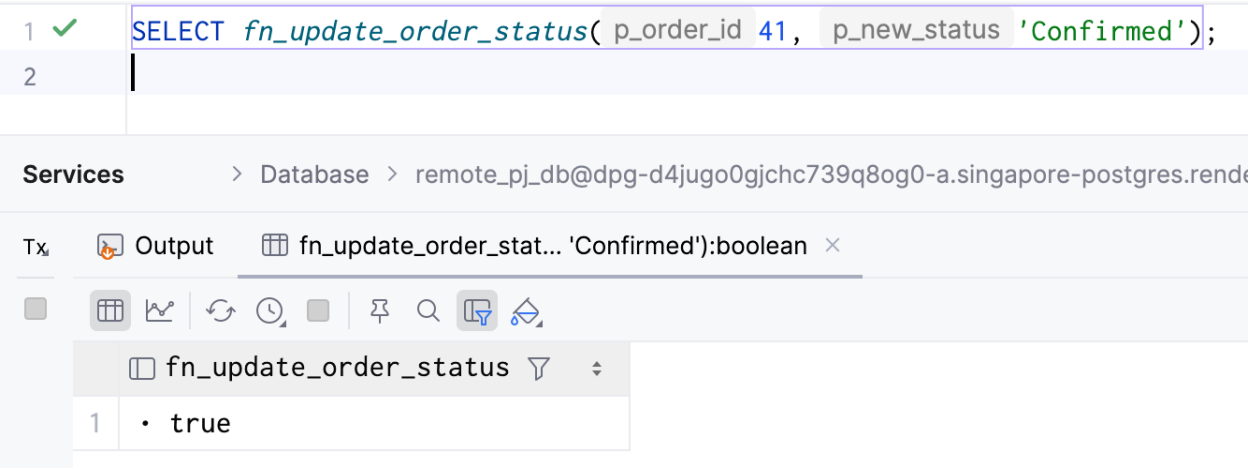


fn_update_order_status

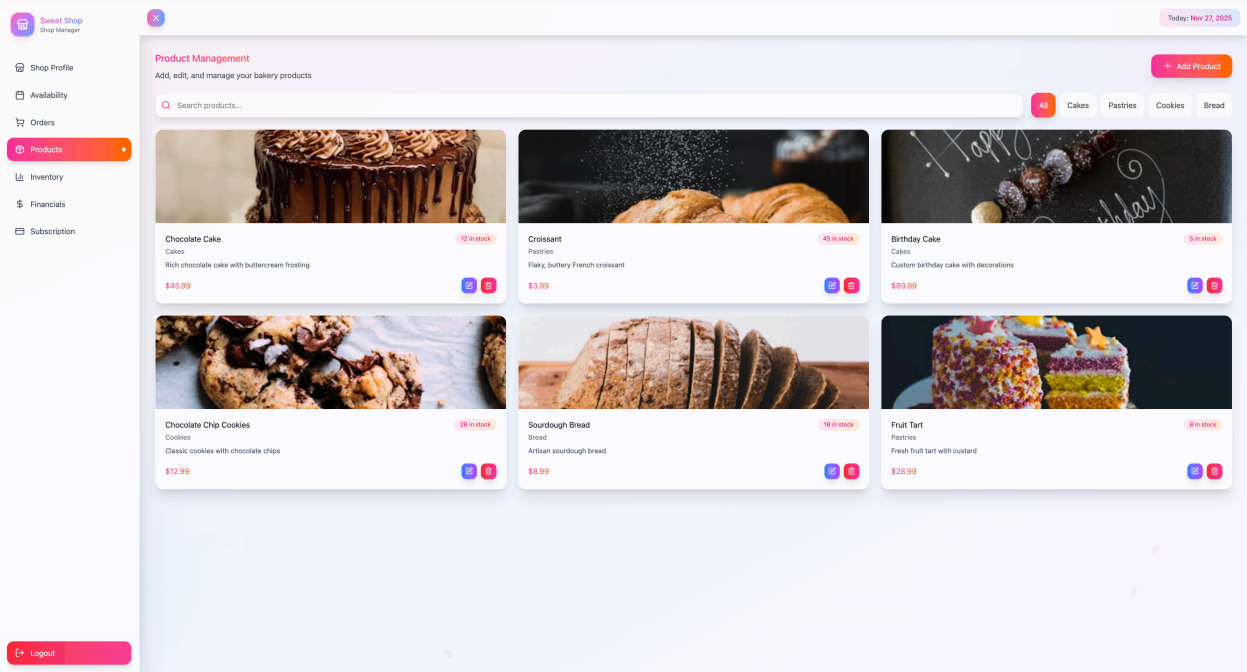
- **Purpose:** Updates an order's status based on workflow progression (e.g., 'Confirmed', 'In Process').
- **How It Works (Logic):** Executes a simple UPDATE on the order table to change the status field.
- **Code:**
- ```
  create function fn_update_order_status(p_order_id integer, p_new_status text)
  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.



Inventory and Product Catalog
  ```



fn_get_shop_products

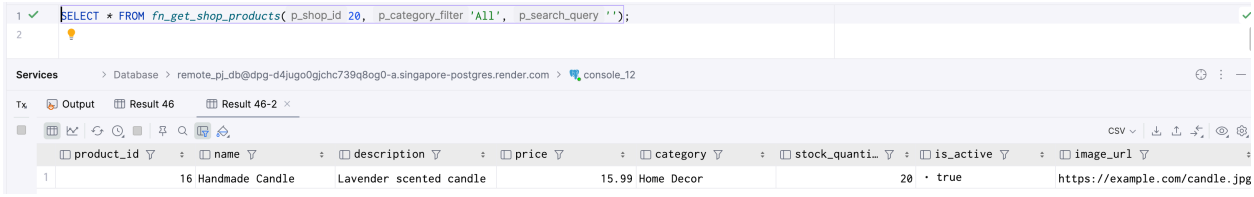
- **Purpose:** Lists the full product catalog for manager editing, including stock information.
- **How It Works (Logic):** Selects detailed product fields filtered by shop_id. Supports filtering by category and searching by name.
- **Code:**

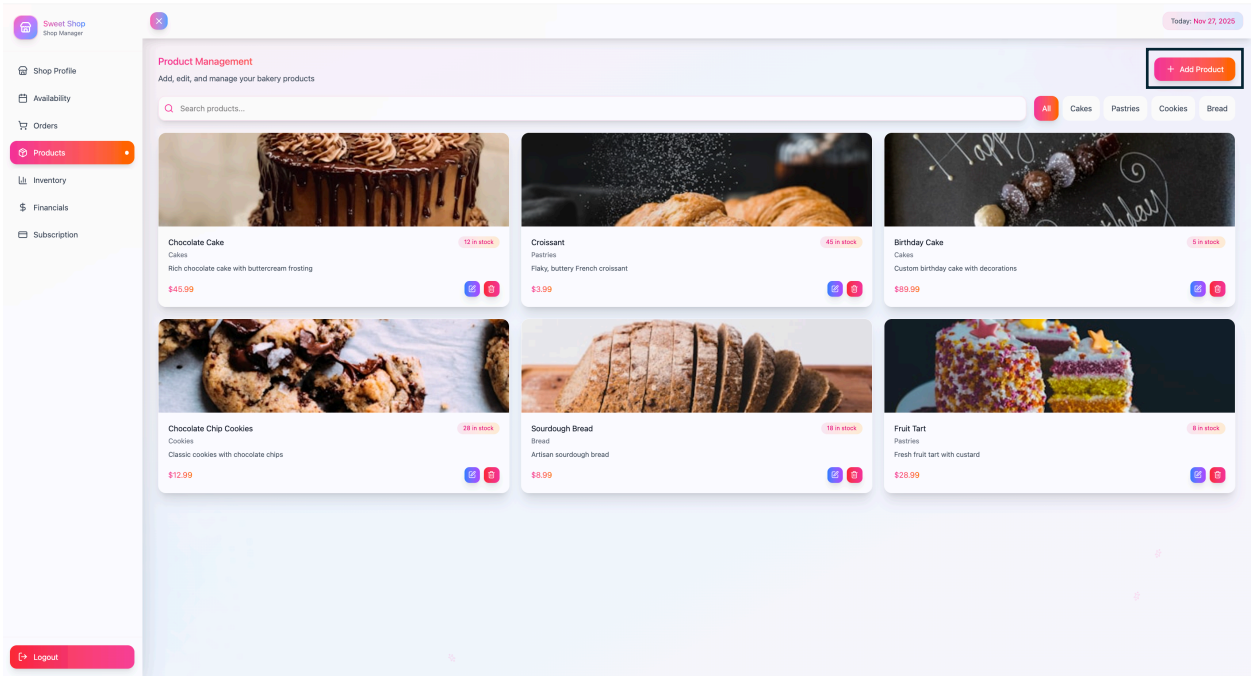
```
create function fn_get_shop_products(p_shop_id integer, p_category_filter text
DEFAULT 'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(product_id integer, name text, description text, price numeric,
category text, stock_quantity integer, is_active boolean, image_url text)
    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
```

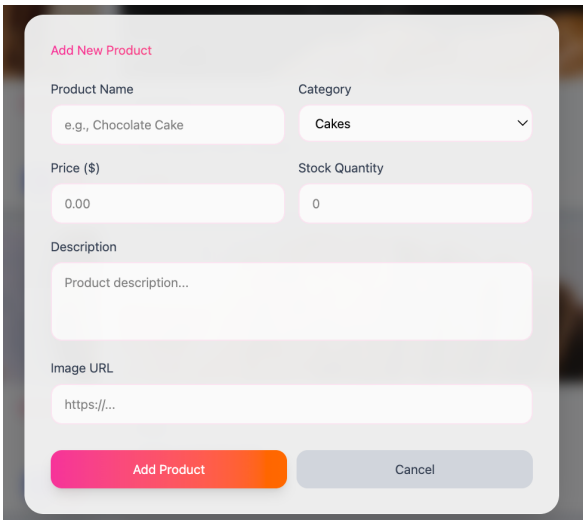
```
            p.shop_id = p_shop_id
        -- Filter by Category (if not 'All')
        AND (p_category_filter = 'All' OR p.category = p_category_filter)
        -- Filter by Search (Product Name)
        AND (
            p_search_query = ''
                OR p.name ILIKE '%' || p_search_query ||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







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.

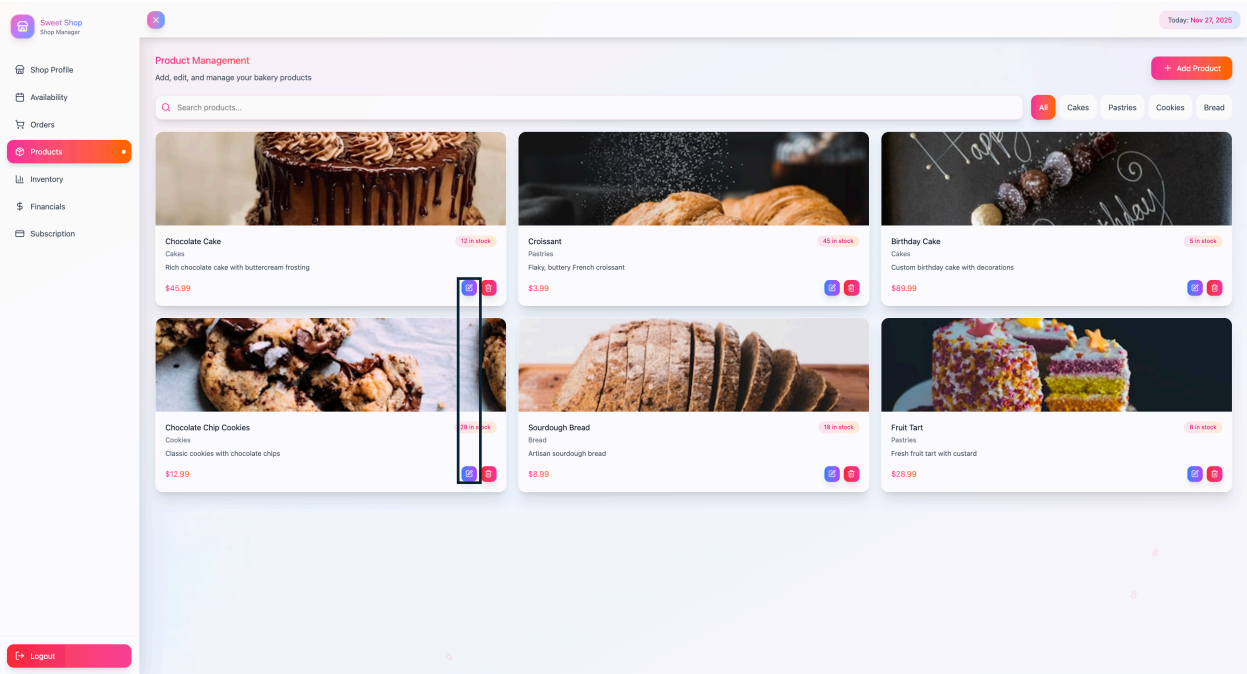
```
SELECT fn_add_shop_product(
        p_shop_id 20, p_name 'Red Velvet Cake', p_description 'Moist red velvet', p_price 35.00, p_category 'Cakes', p_stock_quantity 10, p_image_url 'redvelvet.jpg'
    );
```

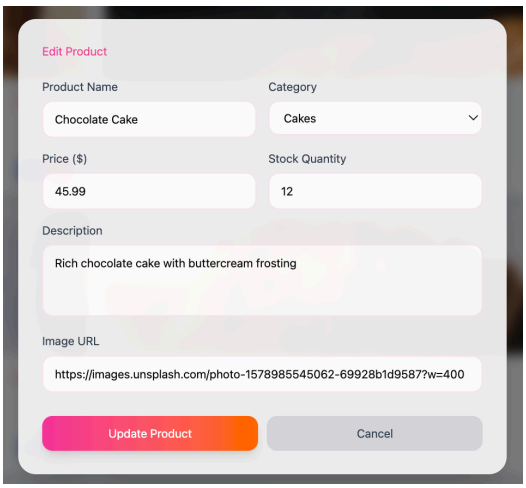
Services > Database > remote_pj_db@dpg-d4jugo0gjchc739q8og0-a.singapore-postgres.render.com > console_12

Output | fn_add_shop_product(...jpg'):integer | Result 46-2

CSV

fn_add_shop_product
18





fn_update_shop_product

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

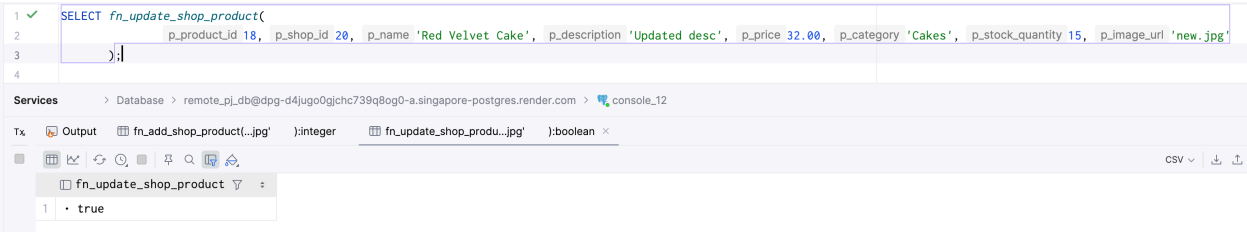
- **How It Works (Logic):** Updates fields in product, setting updated_at = NOW(). Requires product_id and shop_id match.
- **Code:**

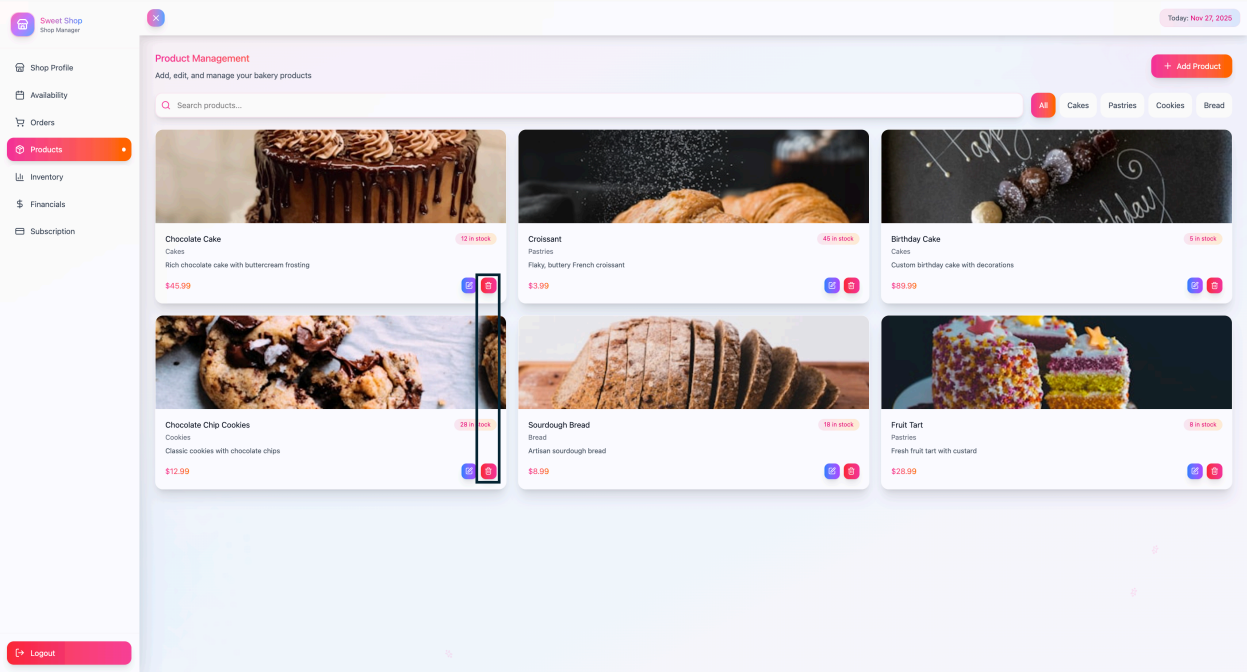
```
create function fn_update_shop_product(p_product_id integer, p_shop_id integer,
p_name text, p_description text, p_price numeric, p_category text,
p_stock_quantity integer, p_image_url text) returns boolean
    language plpgsql
as
$$
BEGIN
    UPDATE product
    SET
        name = p_name,
        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.





fn_delete_shop_product

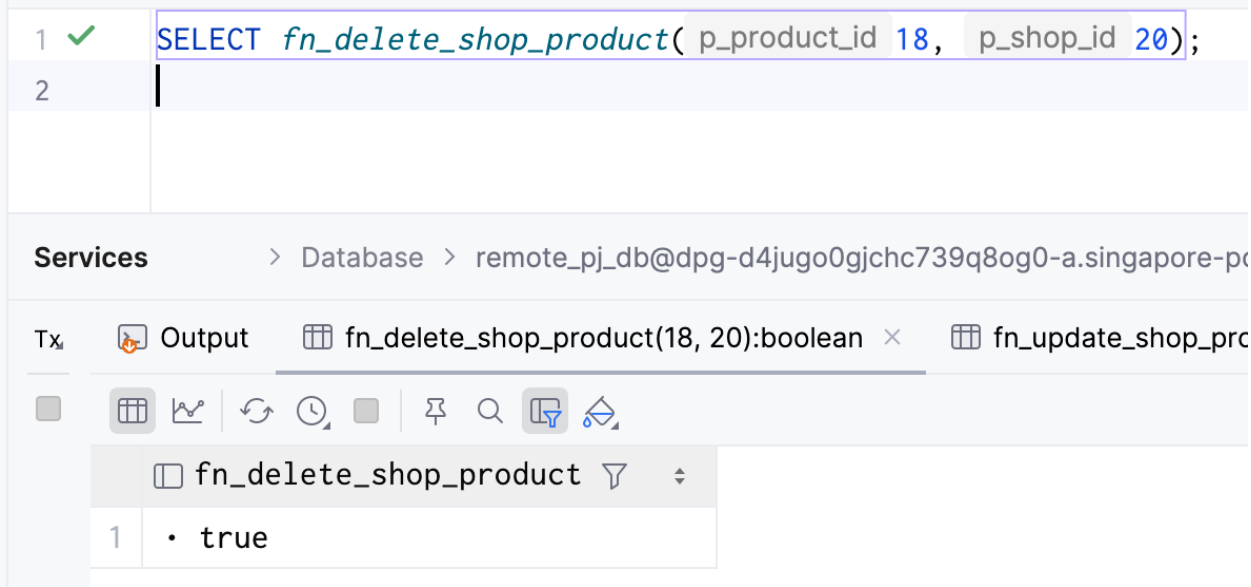
- **Purpose:** Permanently removes a product from the catalog.
- **How It Works (Logic):** Executes DELETE on the product table.
- **Code:**

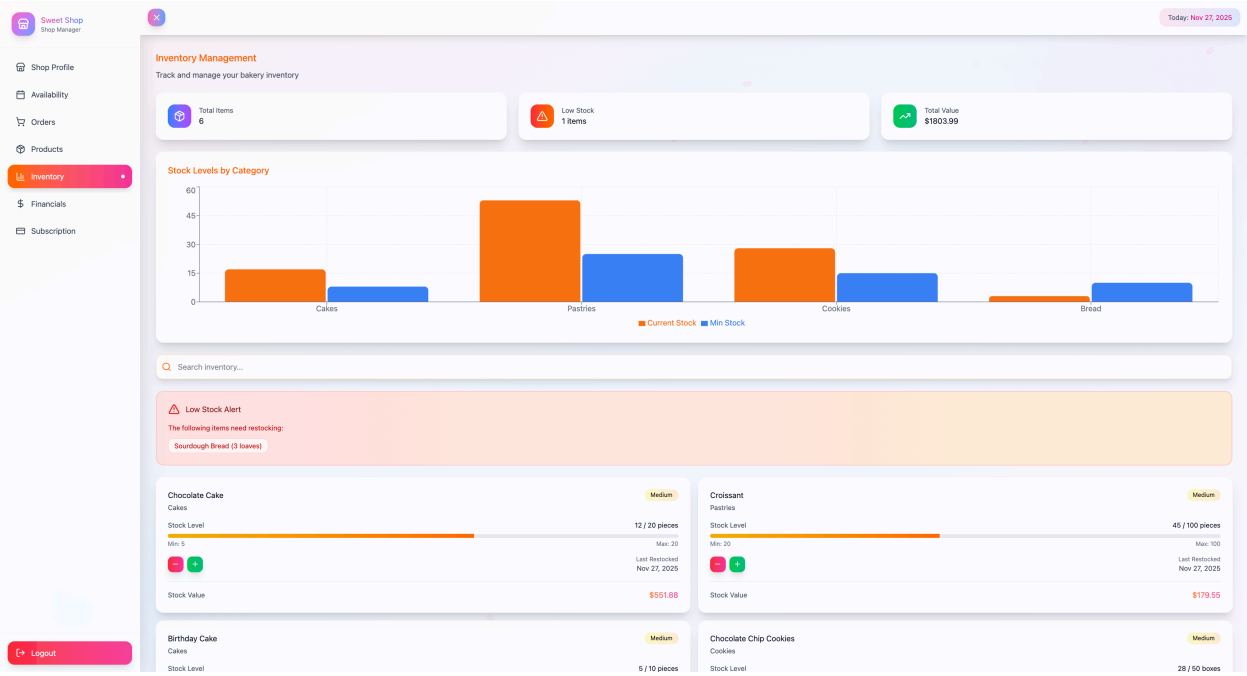
```
create function fn_delete_shop_product(p_product_id integer, p_shop_id integer)
returns boolean
    language plpgsql
as
$$
BEGIN
    DELETE FROM product
    WHERE
        product_id = p_product_id
      AND shop_id = p_shop_id;

    RETURN FOUND;
END;
$$;

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

- **Result Structure:** Returns BOOLEAN.





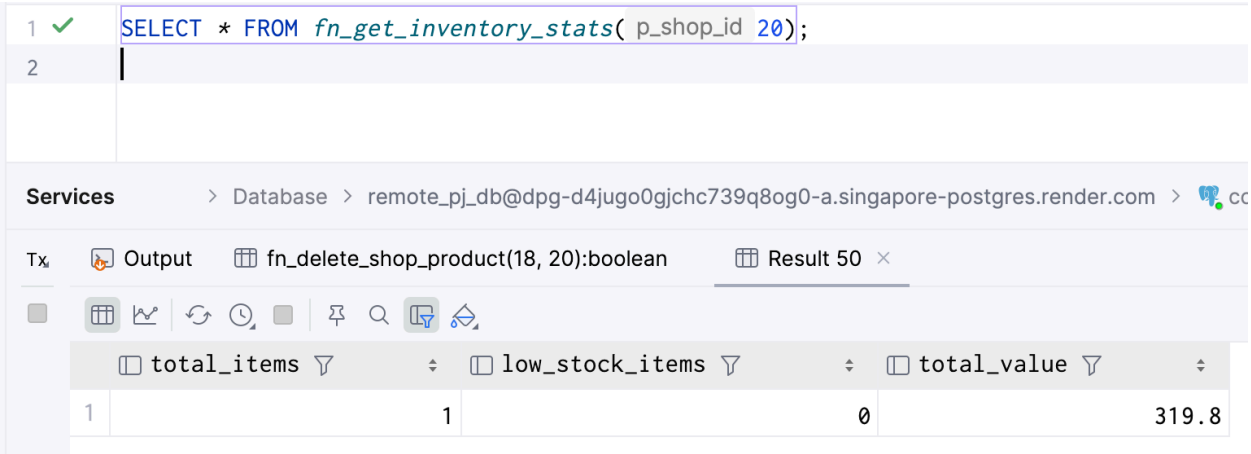
fn_get_inventory_stats

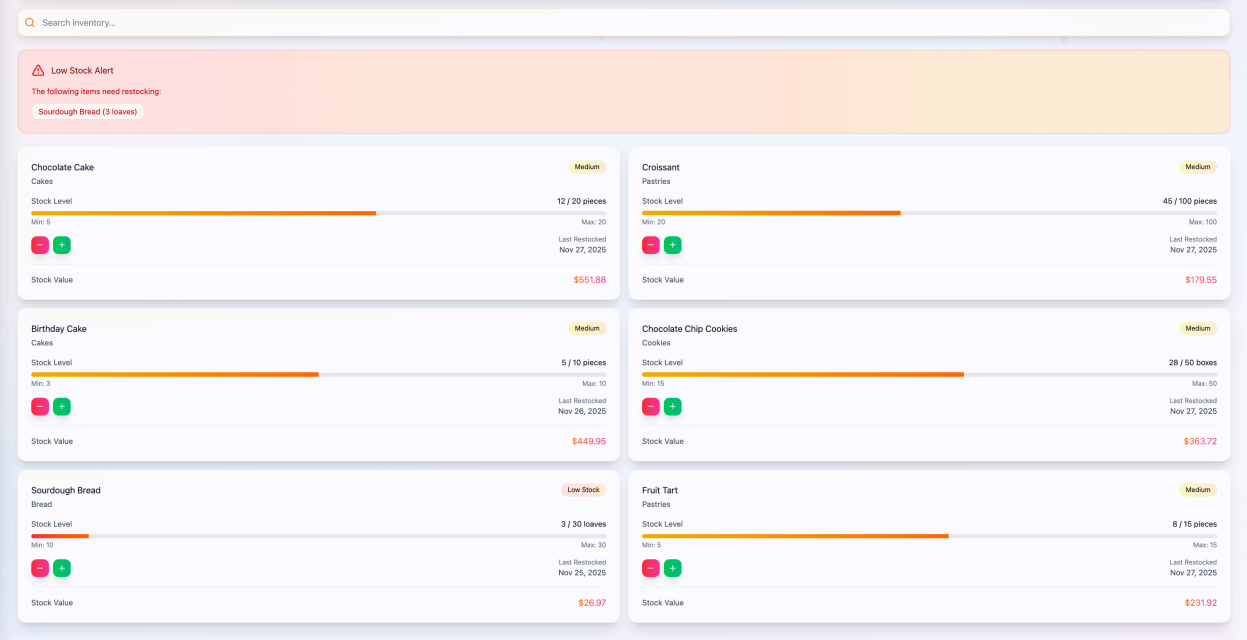
- **Purpose:** Provides summary metrics for overall inventory health.
- **How It Works (Logic):** Aggregates data: COUNT(*) for total items, COUNT(*) FILTER for low stock items, and SUM(stock_quantity * price) for total_value.
- **Code:**

```
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).





fn_get_inventory_list

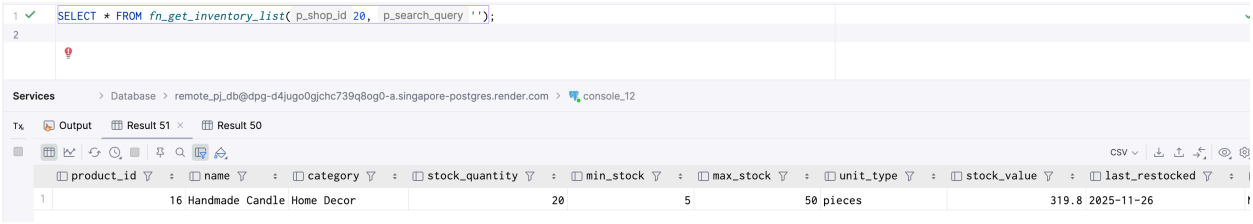
- **Purpose:** Lists inventory items for stock management, highlighting stock status and value.
- **How It Works (Logic):** Selects product data, calculates stock_value, and uses the generated stock_status column. Orders results to prioritize low-stock items first.
- **Code:**

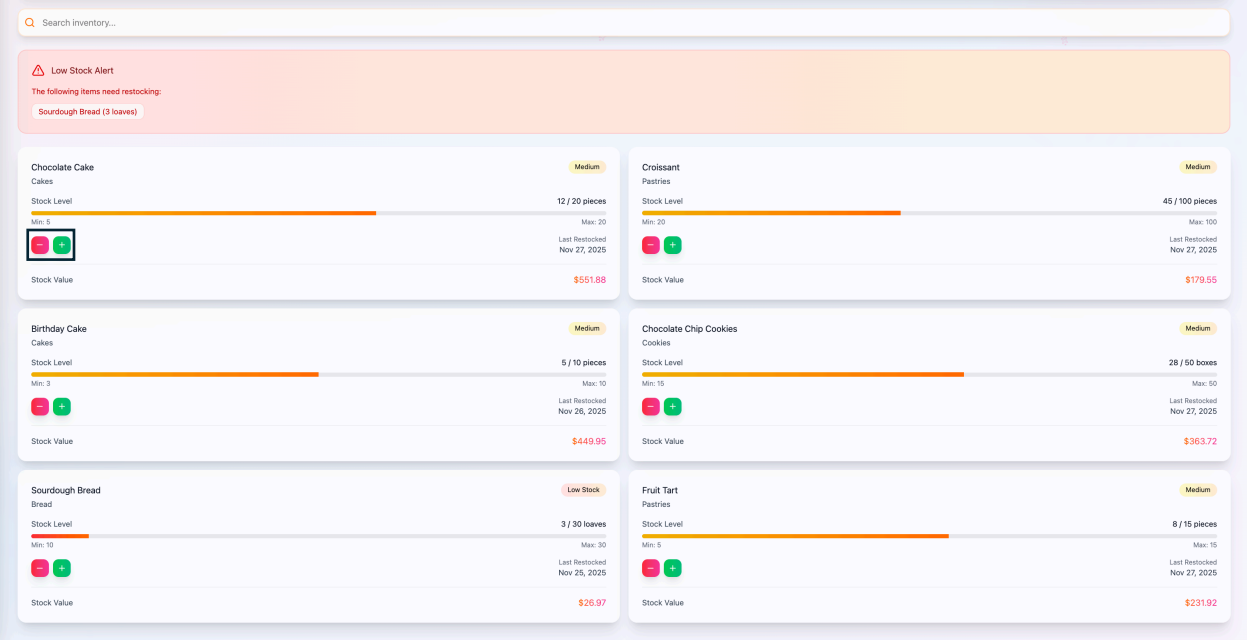
```
create function fn_get_inventory_list(p_shop_id integer, p_search_query text
DEFAULT ''::text)
    returns TABLE(product_id integer, name text, category text, stock_quantity
integer, min_stock integer, max_stock integer, unit_type text, stock_value
numeric, last_restocked date, status text)
    language 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).





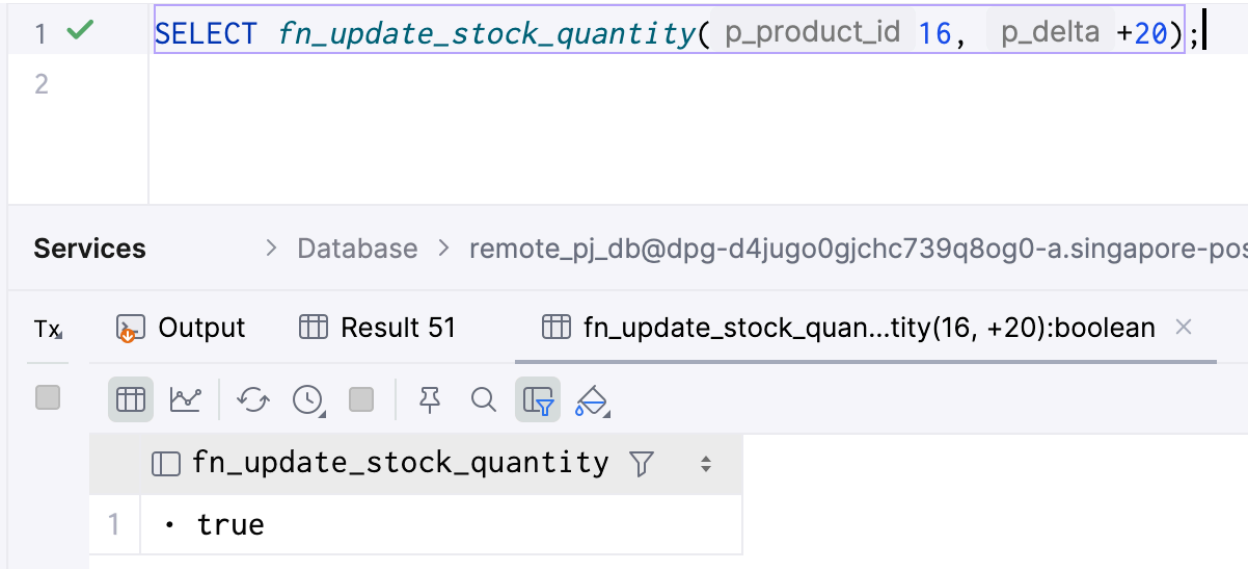
fn_update_stock_quantity

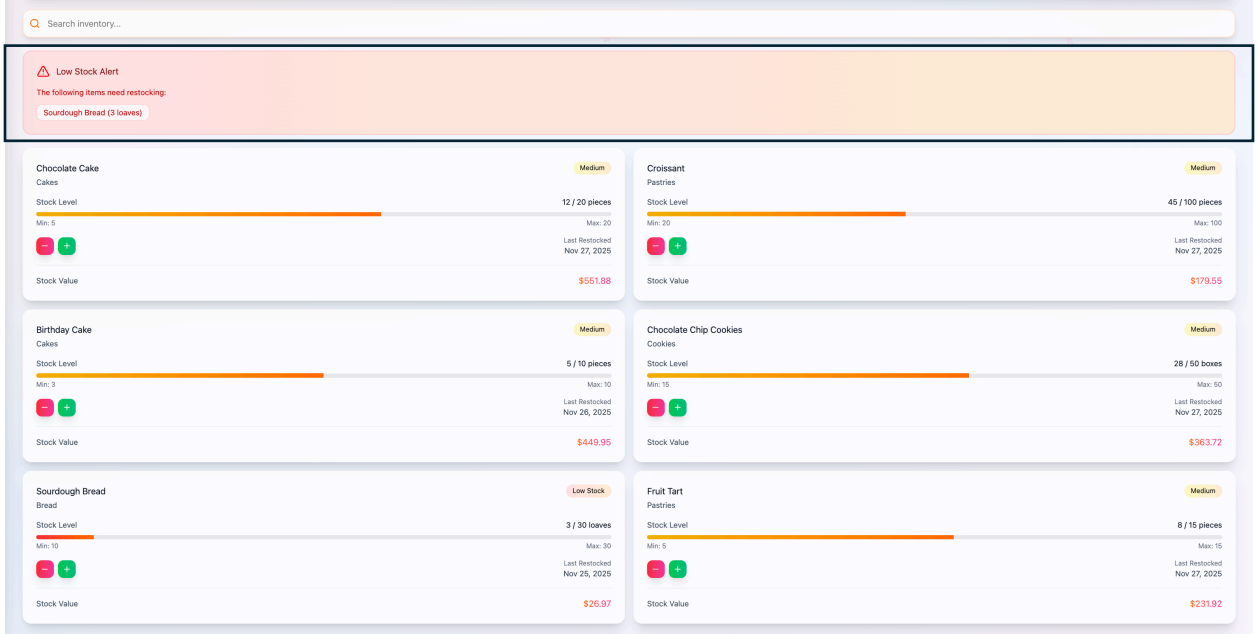
- **Purpose:** Adjusts the stock level of a product.
- **How It Works (Logic):** Updates stock_quantity by adding p_delta, using GREATEST(0, ...) to prevent negative stock. Updates last_restocked_at only if stock was added (p_delta > 0).
- **Code:**
- ```
  create function fn_update_stock_quantity(p_product_id 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.





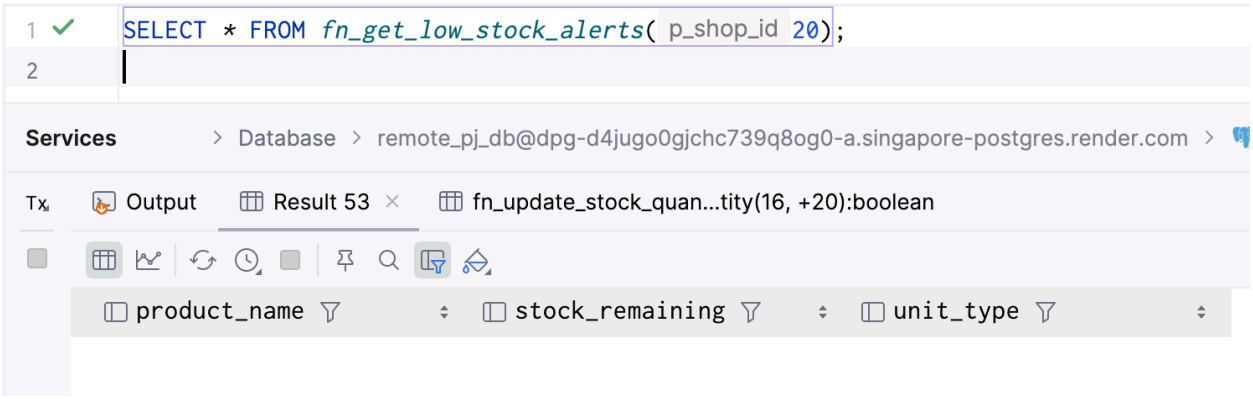
fn_get_low_stock_alerts

- **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:**

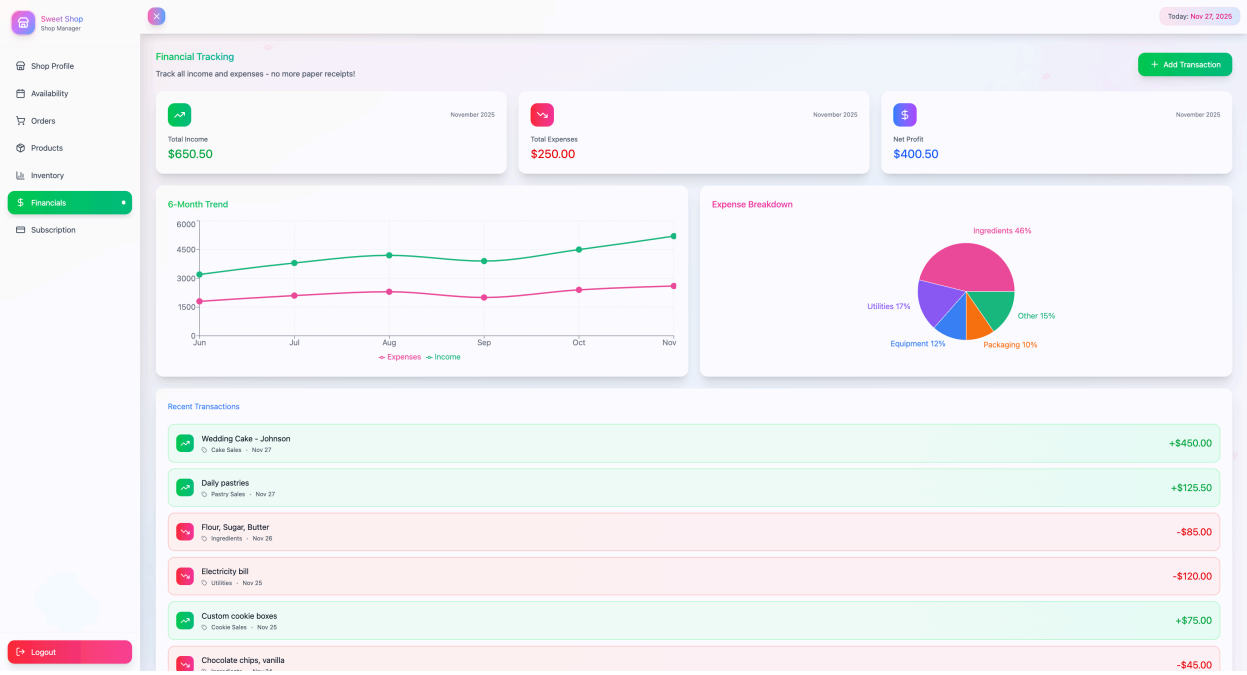
```
create function fn_get_low_stock_alerts(p_shop_id integer)
    returns TABLE(product_name text, stock_remaining integer, unit_type text)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            p.name,
            p.stock_quantity,
            p.unit_type
        FROM
       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

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

- **Result Structure:** Returns alerts: (product_name TEXT, stock_remaining INTEGER, unit_type TEXT).



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:**

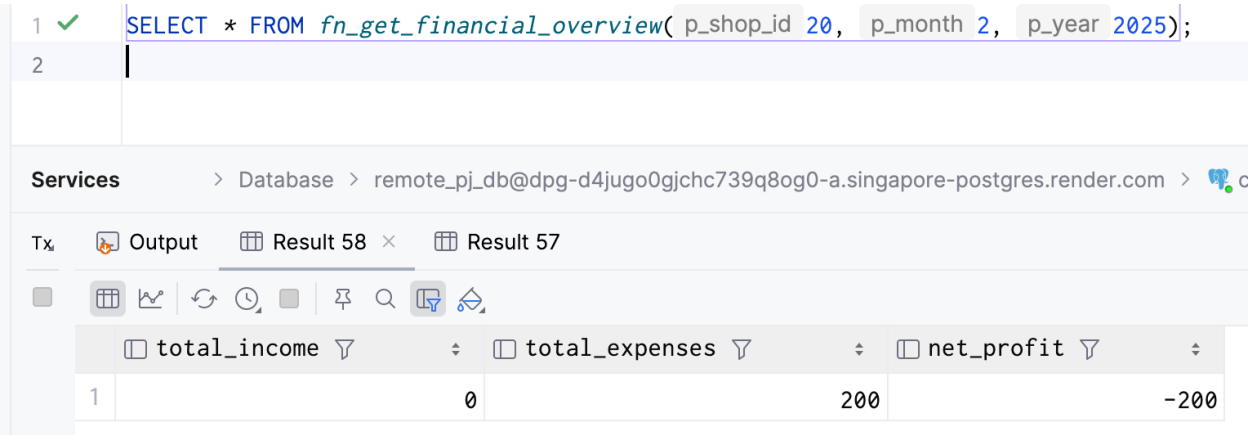
```
create function fn_get_financial_overview(p_shop_id integer, p_month integer,
p_year integer)
    returns TABLE(total_income numeric, total_expenses numeric, net_profit
numeric)
    language plpgsql
as
$$
DECLARE
    v_income DECIMAL(10,2);
    v_expenses DECIMAL(10,2);
BEGIN
    -- Calculate Income
    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;
```

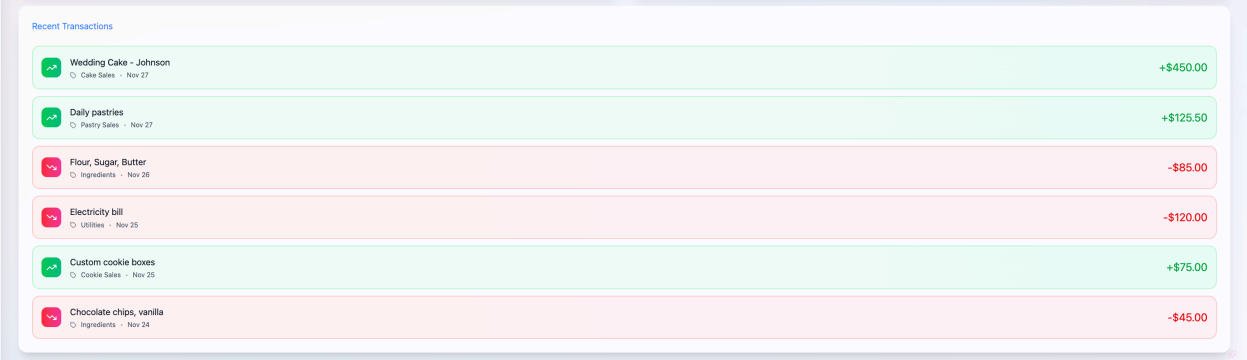
```
    -- Calculate Expenses
    SELECT COALESCE(SUM(amount), 0.00) INTO v_expenses
    FROM shop_transaction
    WHERE shop_id = p_shop_id
      AND type = 'Expense'
      AND EXTRACT(MONTH FROM transaction_date) = p_month
      AND EXTRACT(YEAR FROM transaction_date) = p_year;

    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).





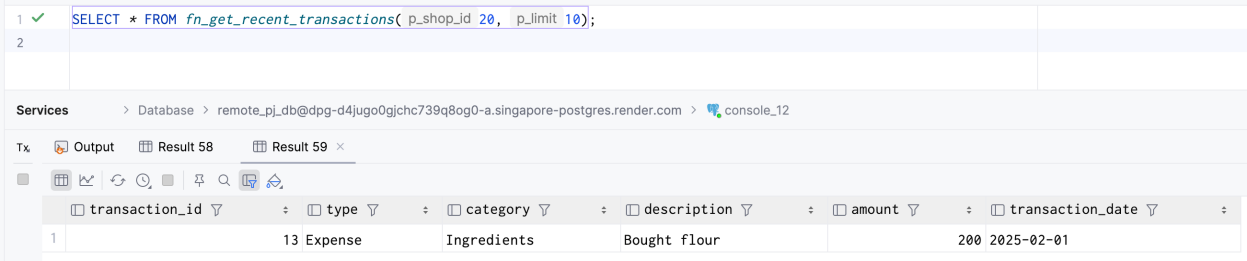
fn_get_recent_transactions

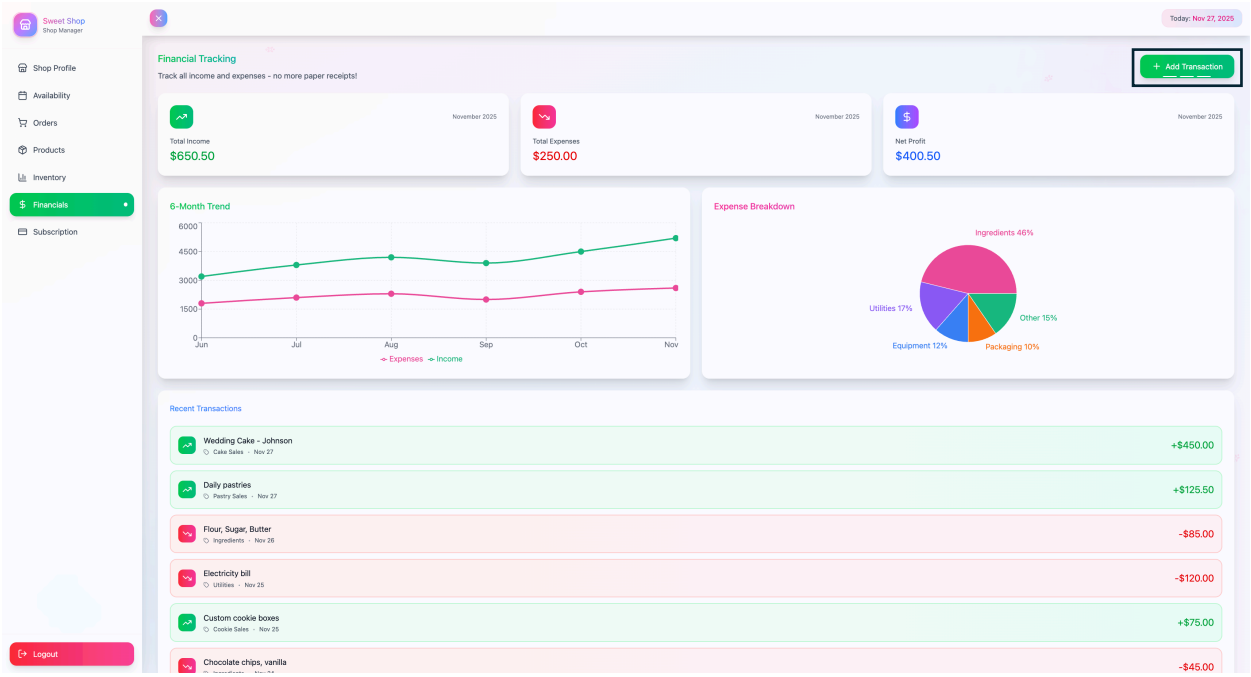
- **Purpose:** Retrieves the shop's most recent financial transactions for a dashboard feed.
- **How It Works (Logic):** Queries shop_transaction, ordered by date and creation time descending, limited by p_limit.
- **Code:**

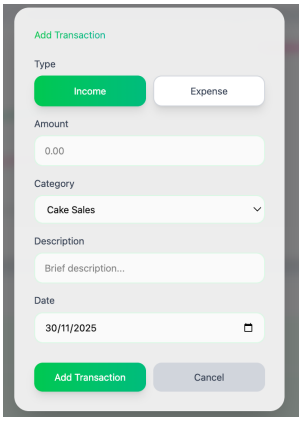
```
create function fn_get_recent_transactions(p_shop_id integer, p_limit integer
DEFAULT 5)
    returns TABLE(transaction_id integer, type text, category text, description
text, amount numeric, transaction_date date)
    language plpgsql
as
$$
BEGIN
    RETURN QUERY
        SELECT
            t.transaction_id,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

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).







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:**

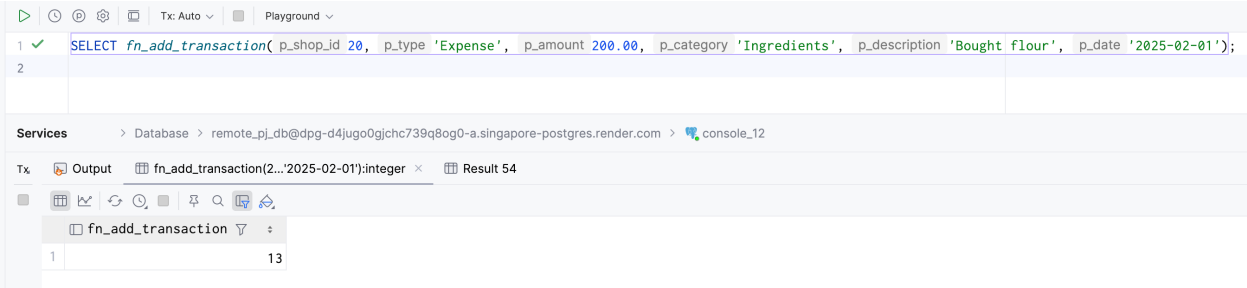
```
create function fn_add_transaction(p_shop_id integer, p_type text, p_amount
numeric, p_category text, p_description text, p_date date DEFAULT CURRENT_DATE)
returns integer
    language plpgsql
as
$$
DECLARE
    v_new_id INT;
BEGIN
    INSERT INTO shop_transaction (
```

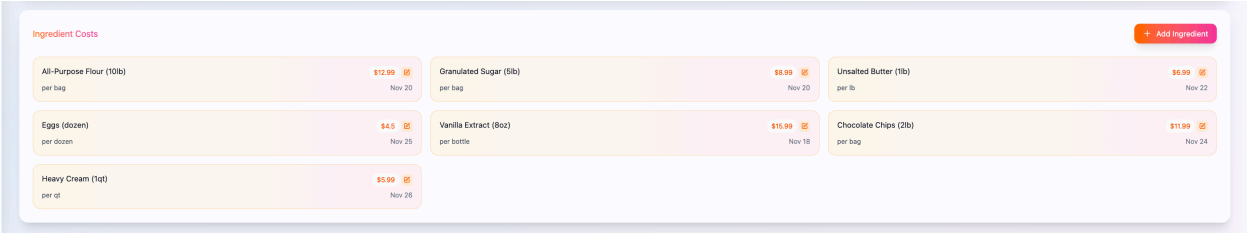
```
        shop_id, type, amount, category, description, transaction_date
    )
    VALUES (
                p_shop_id, p_type, p_amount, p_category, p_description, p_date
            )
    RETURNING transaction_id INTO v_new_id;

    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.





fn_get_shop_ingredients

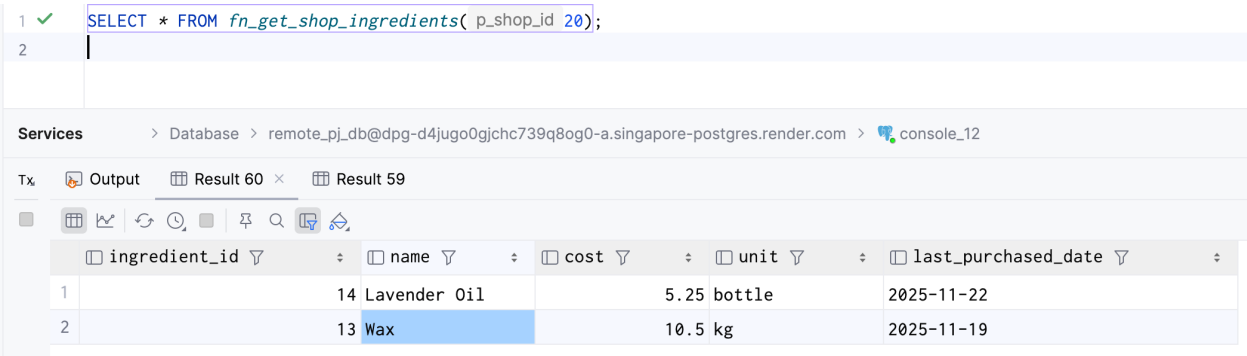
- **Purpose:** Retrieves the list of raw materials for COGS tracking.
- **How It Works (Logic):** Selects ingredient data filtered by shop_id.
- **Code:**

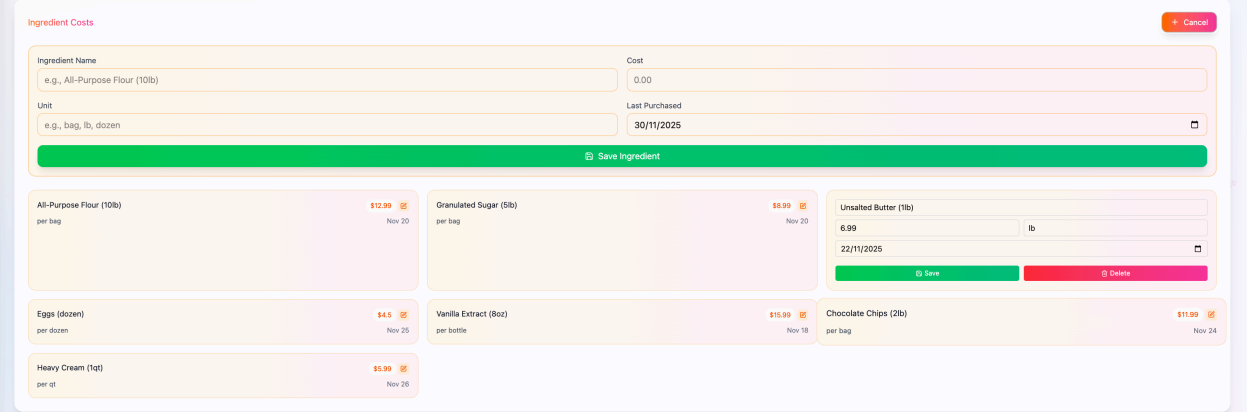
```
create function fn_get_shop_ingredients(p_shop_id integer)
    returns TABLE(ingredient_id integer, name text, cost numeric, unit text,
last_purchased_date date)
    language plpgsql
as
$$
BEGIN
```

```
    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).





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:**

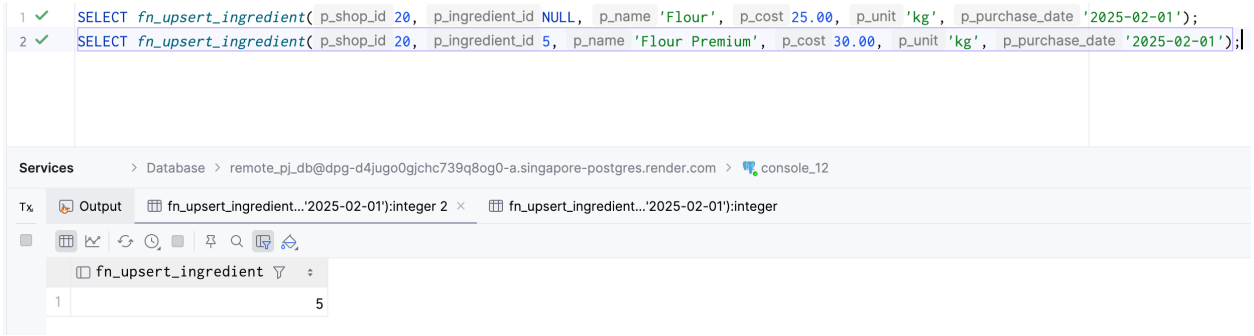
```
create function fn_upsert_ingredient(p_shop_id integer, p_ingredient_id integer,
p_name text, p_cost numeric, p_unit text, p_purchase_date date) returns integer
    language plpgsql
as
$$
DECLARE
    v_id INT;
BEGIN
    IF p_ingredient_id IS NULL THEN
        -- Create
        INSERT INTO shop_ingredient (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;

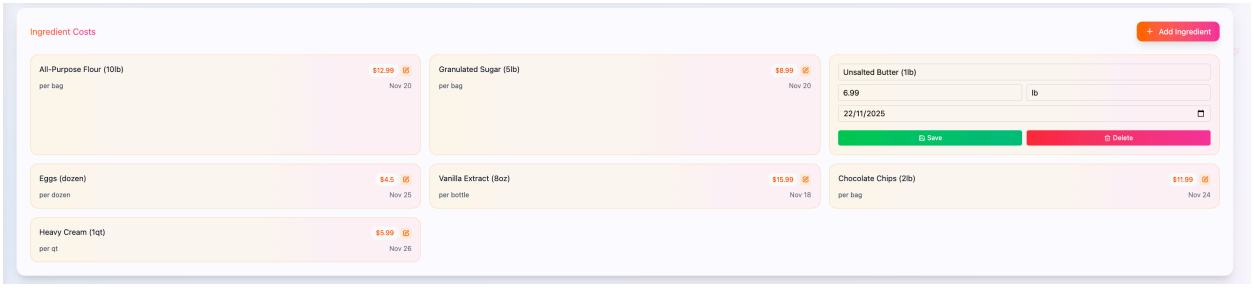
    RETURN v_id;
```

```
END;
$$;

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

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





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;

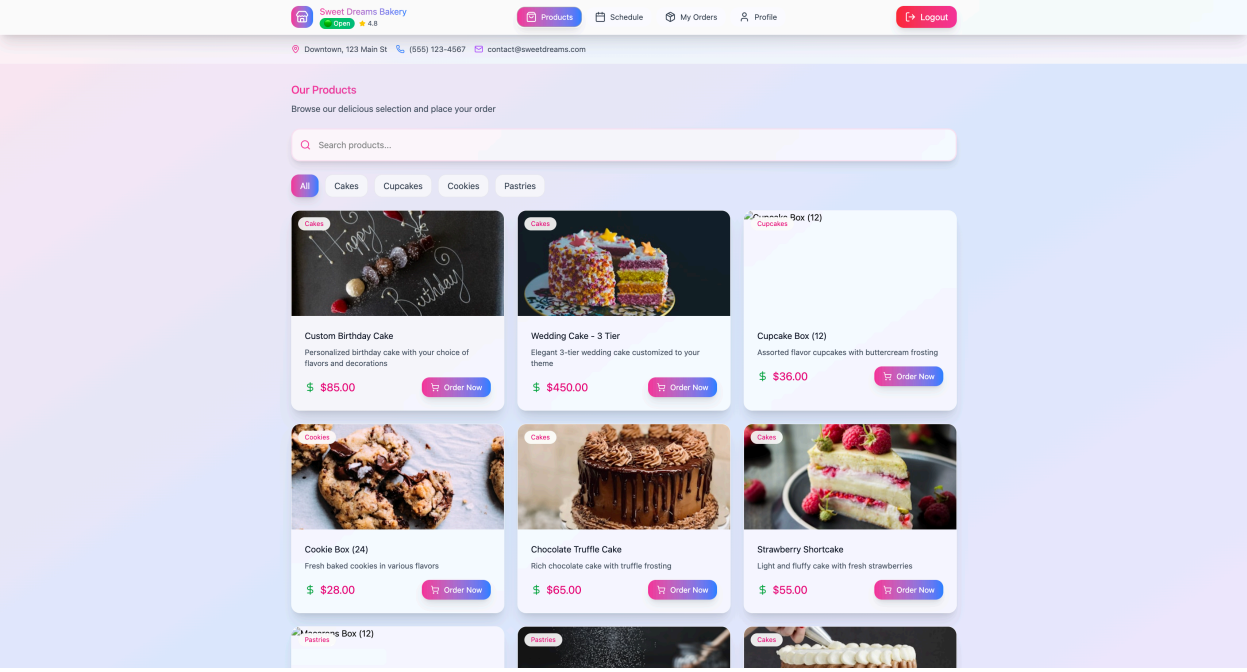
    RETURN FOUND;
```

```
END;
$$;

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

- **Result Structure:** Returns BOOLEAN.

c. Customer-Facing Functions



fn_get_customer_products

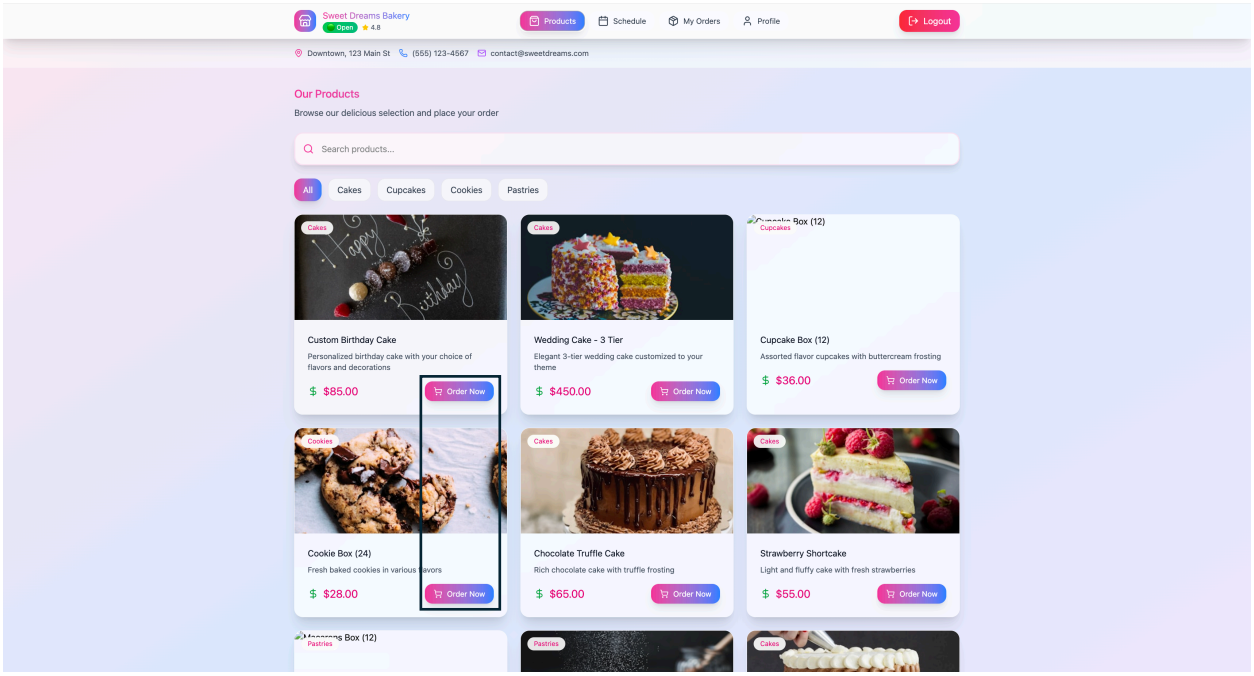
- **Purpose:** Lists products available to customers on the public store front.
- **How It Works (Logic):** Filters the product catalog by is_active = TRUE and explicitly enforces **stock_quantity > 0** (only showing items available to buy). Supports category and search filtering.
- **Code:**

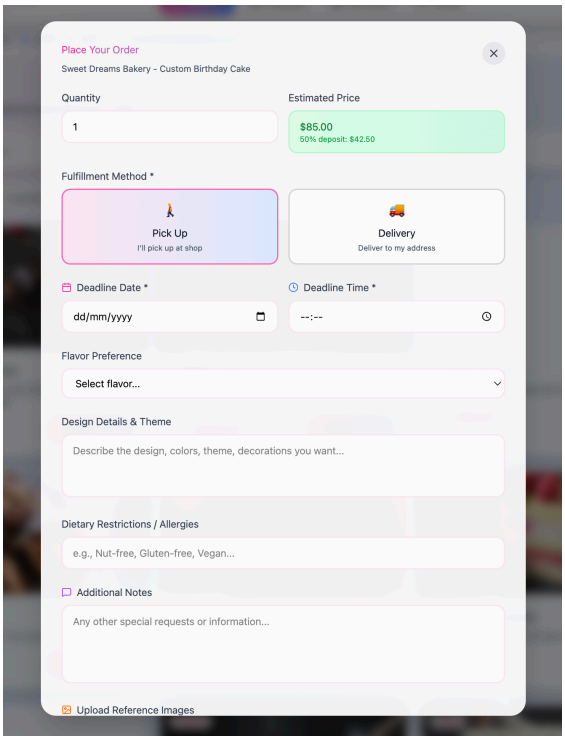
```
create function fn_get_customer_products(p_shop_id integer, p_category_filter text
DEFAULT 'All'::text, p_search_query text DEFAULT ''::text)
    returns TABLE(product_id integer, name text, description text, price numeric,
category text, image_url text)
    language plpgsql
as
$$
BEGIN
    RETURN 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).





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;
$$;

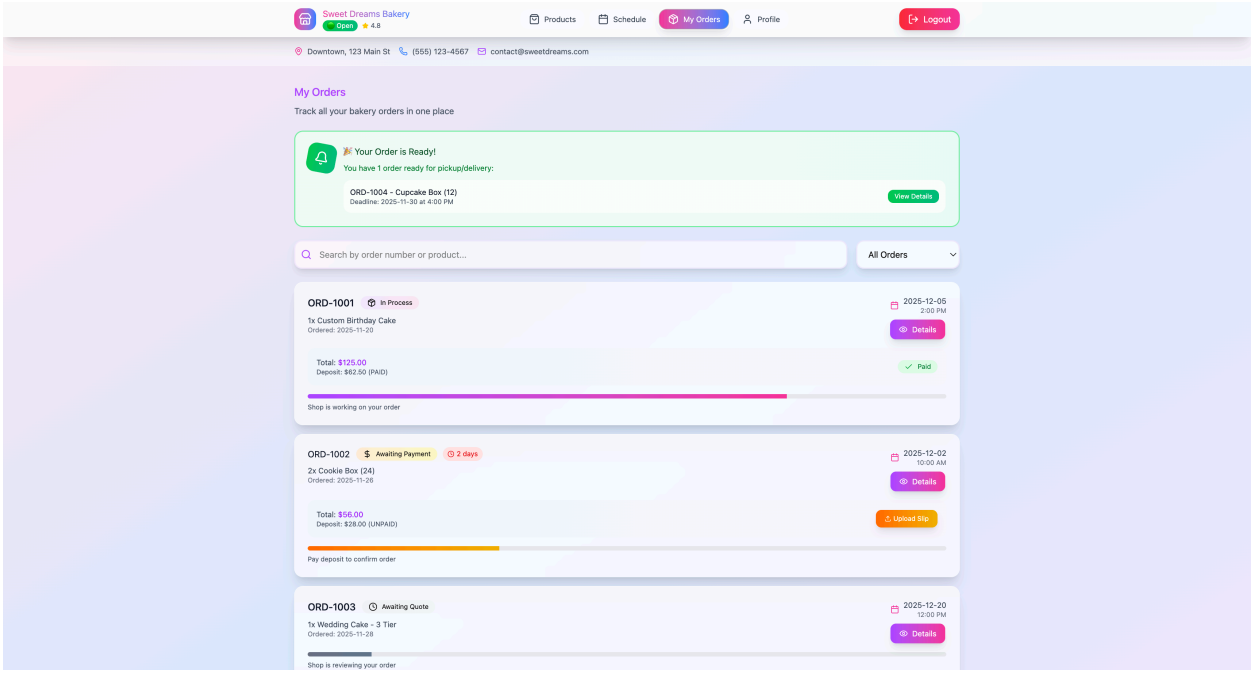
alter function fn_submit_customer_order(integer, integer, text, integer, date,
time, text, text, text, text) owner to root;
```

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

```
SELECT fn_submit_customer_order(
                p_shop_id 20, p_customer_id 23,
                p_product_name 'Chocolate Cake', p_quantity 2,
                p_deadline '2025-02-15', p_deadline_time '14:00',
                p_fulfillment_type 'Pickup',
                p_flavor 'Chocolate', p_design_notes 'Birthday theme', p_dietary_notes 'No nuts'
        );
```

Output | fn_submit_customer_o...uts' |):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
  $$
  BEGIN
      RETURN QUERY
          SELECT
              o.order_id,
              COALESCE(
                      (SELECT string_agg(quantity || 'x ' || product_name, ', ')
                       FROM order_item oi WHERE oi.order_id = o.order_id),
                      'Custom Order'
              )::TEXT AS product_summary,
              o.status,
              o.total_amount,
              o.deposit_amount,
              o.deposit_status,
              o.deadline,
              o.created_at,
              (o.status = 'Ready for Pickup' OR o.status = 'Ready for Delivery') AS
  is_ready
          FROM
              "order" o
          WHERE
              o.customer_id = p_customer_id
            AND (p_status_filter = 'All' OR o.status = p_status_filter)
            AND (
              p_search_query = ''
                  OR o.order_id::TEXT ILIKE '%' || p_search_query || '%'
                  OR EXISTS (
                  SELECT 1 FROM order_item oi
                  WHERE oi.order_id = o.order_id
                    AND oi.product_name ILIKE '%' || p_search_query || '%'
              )
              )
          ORDER BY
              o.created_at DESC;
  END;
  $$;

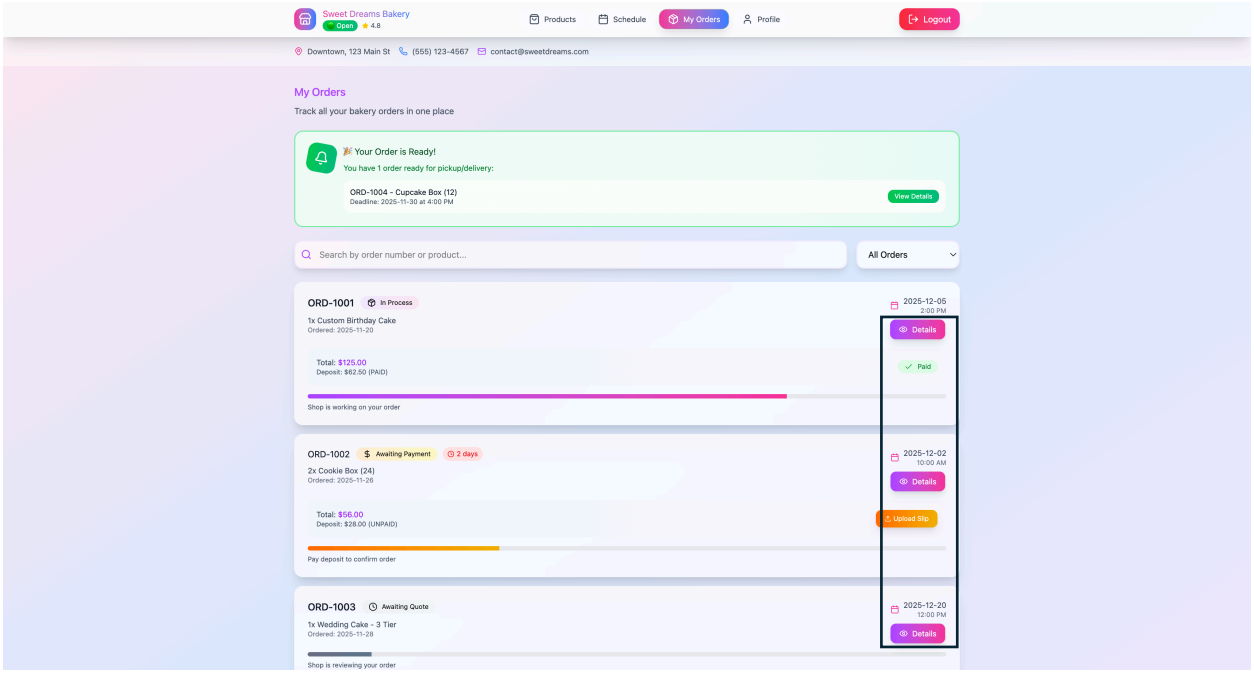
  alter function fn_get_customer_orders(integer, text, text) owner to root;
  ```

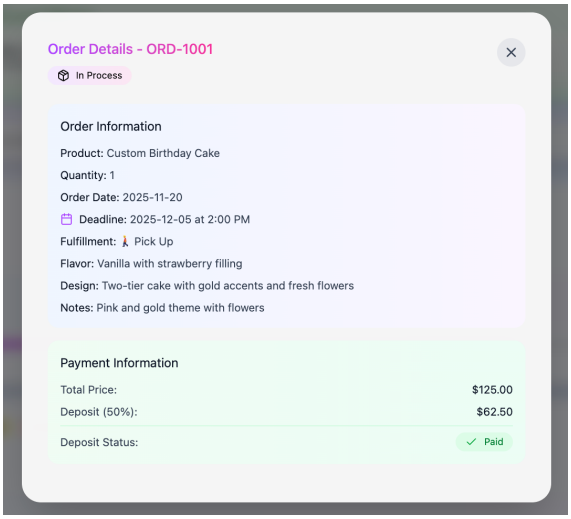
- **Result Structure:** Returns the customer's order list: (order_id INTEGER, product_summary TEXT, status TEXT, total_amount NUMERIC, deposit_amount NUMERIC, deposit_status TEXT, deadline TIMESTAMP WITH TIME ZONE, created_at TIMESTAMP WITH TIME ZONE, is_ready BOOLEAN).
  ```

```
SELECT * FROM fn_get_customer_orders( p_customer_id 23, p_status_filter 'All', p_search_query '');
```

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 14:00:00.0





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;
```

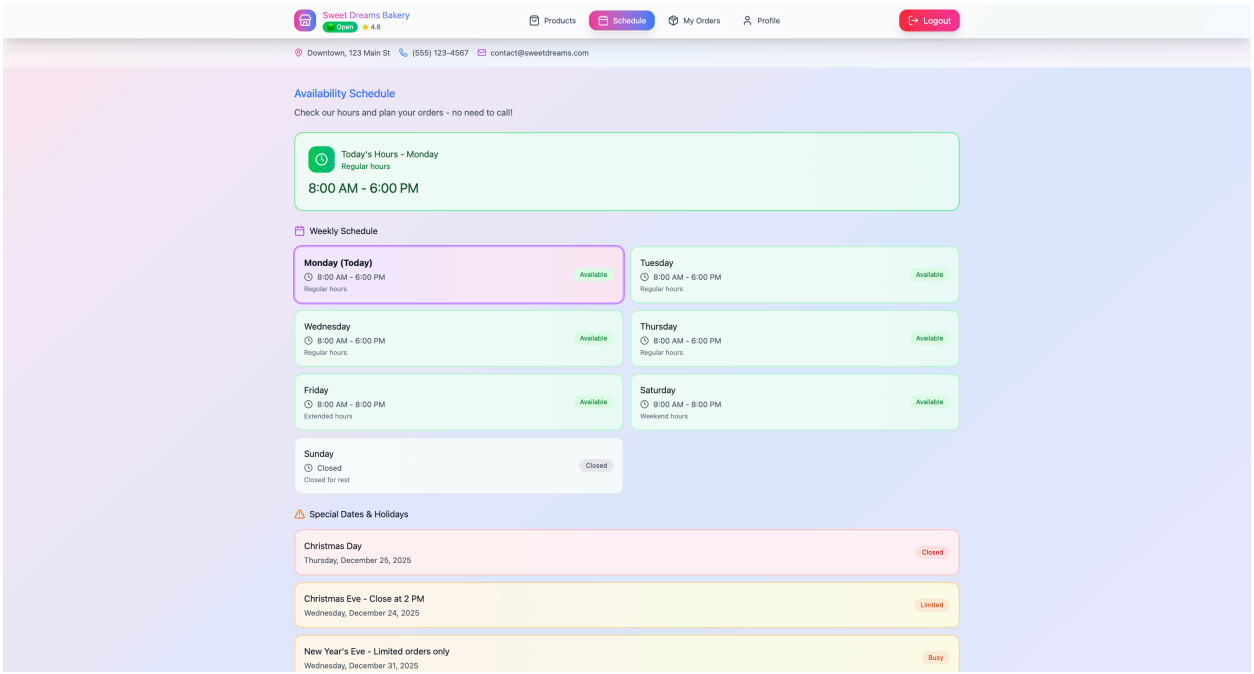
- **Result Structure:** Returns the full details of one order: (order_id INTEGER, status TEXT, product_name TEXT, quantity INTEGER, order_date DATE, deadline TIMESTAMP WITH TIME ZONE, fulfillment_method TEXT, delivery_address TEXT,

item_notes TEXT, total_price NUMERIC, deposit_amount NUMERIC, deposit_status TEXT, payment_slip_url TEXT).

```
SELECT * FROM fn_get_customer_order_details( p_order_id 42,  p_customer_id 23);
```

Output | Result 79

	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 +00:0	<null>



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;
$$;

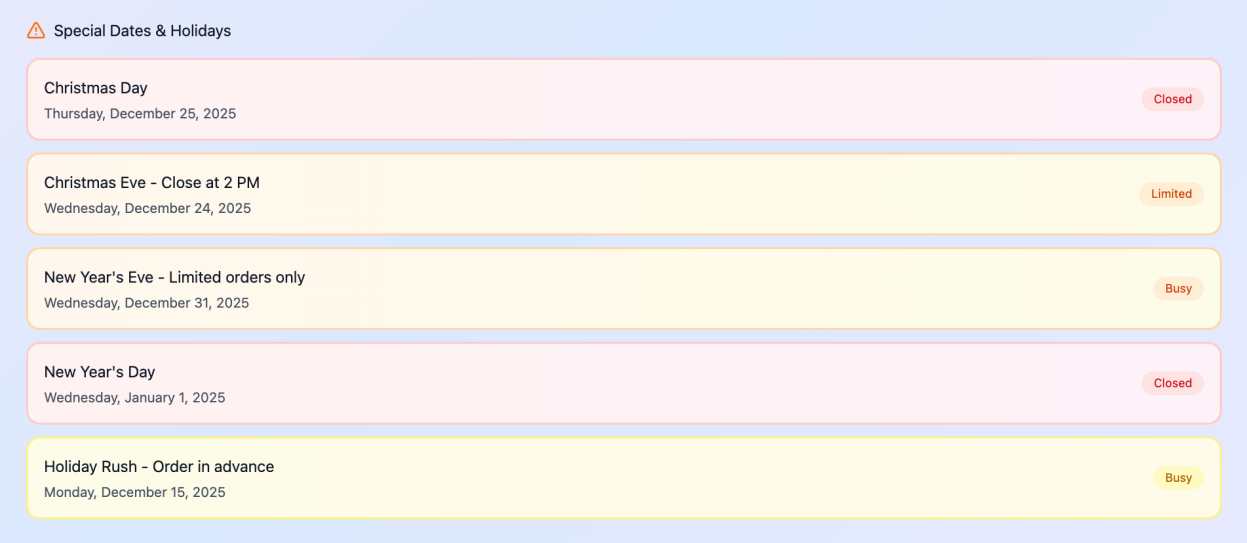
alter function fn_get_customer_weekly_schedule(integer) owner to root;
```

- **Result Structure:** Returns the formatted weekly schedule: (day_of_week TEXT, time_range TEXT, status_tag TEXT, is_today BOOLEAN).

```
SELECT * FROM fn_get_customer_weekly_schedule( p_shop_id 20);
```

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



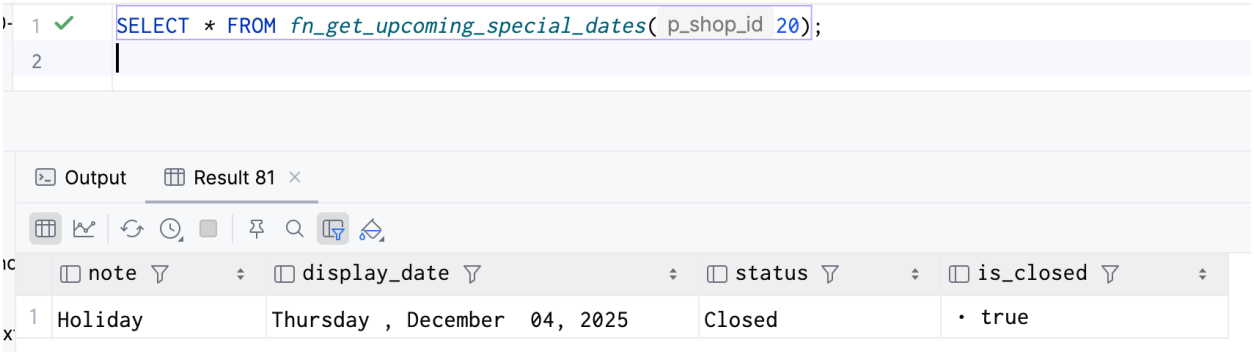
fn_get_upcoming_special_dates

- **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).



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