Online Store Name

- **TechNest**

Product or Service Description

- TechNest is an online store that offers the latest gadgets and tech accessories from smartwatches, wireless earbuds, and gaming peripherals to phone accessories and home tech gear. We aim to be a one-stop-shop for all things tech.

Target Customers

- Our primary customers are **tech enthusiasts**, including students, gamers, young professionals, and early adopters of new technology.

Key Features

Customer View:

- Product Search and Filtering
- Browse Product Catalog
- User Login and Registration
- Mock Checkout Process (no real payment)

Admin View:

- User Management Dashboard

Staff View:

- Inventory Stock Management Access

Team Members

- Paolo Mendoza – Frontend Developer
- Ej Sadiarin – Backend Developer
- Jennilyn Ching – Database Administrator
- Marius Manaloto – UI/UX Designer

Tech Stack

- **Frontend:** React
- **Backend:** TypeScript
- **Database:** MySQL

DB Models

1. users
2. categories
3. products
4. inventory
5. order
6. order_items

1. `users` Table

- Stores information about registered users, including both customers and administrators.

Column Name	Data Type	Constraints	Description
`user_id`	`INT`	`PRIMARY KEY`, `AUTO_INCREMENT`	Unique identifier for each user
`username`	`VARCHAR(255)`	`NOT NULL`, `UNIQUE`	User's chosen username
`email`	`VARCHAR(255)`	`NOT NULL`, `UNIQUE`	User's email address
`password_hash`	`VARCHAR(255)`	`NOT NULL`	Hashed password for security
`first_name`	`VARCHAR(255)`		User's first name
`last_name`	`VARCHAR(255)`		User's last name
`address`	`TEXT`		User's shipping address
`phone_number`	`VARCHAR(20)`		User's phone number
`role`	`ENUM('customer', 'admin', ‘staff’)`	`NOT NULL`, `DEFAULT 'customer'`	User's role (customer or admin)

created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the user account was created
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	Timestamp of last update to user account

2. `categories` Table

Organizes products into different categories for easier browsing and filtering.

Column Name	Data Type	Constraints	Description
category_id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each category
name	VARCHAR(255)	NOT NULL, UNIQUE	Name of the category (e.g., "Smartwatches", "Gaming Peripherals")
description	TEXT		Optional description of the category

3. `products` Table

Stores details about each product available in the store.

Column Name	Data Type	Constraints	Description
product_id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each product
name	VARCHAR(255)	NOT NULL	Name of the product

description	TEXT		Detailed description of the product
price	DECIMAL(10, 2)	NOT NULL, CHECK (price >= 0)	Price of the product
category_id	INT	NOT NULL, FOREIGN KEY REFERENCES categories(category_id)	ID of the product's category
image_url	VARCHAR(255)		URL to the product's main image
brand	VARCHAR(255)		Brand of the product
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Timestamp when the product was added
updated_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	Timestamp of last update to product details

4. `inventory` Table

Manages the stock levels for each product.

Column Name	Data Type	Constraints	Description
inventory_id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each inventory record
product_id	INT	NOT NULL, UNIQUE, FOREIGN KEY REFERENCES products(product_id)	ID of the product
stock_quantity	INT	NOT NULL, DEFAULT 0, CHECK (stock_quantity >= 0)	Current quantity of the product in stock

last_updated	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP	Timestamp of the last stock update

5. `orders` Table

Stores information about customer orders.

Column Name	Data Type	Constraints	Description
order_id	INT	PRIMARY KEY, AUTO_INCREMENT	Unique identifier for each order
user_id	INT	NOT NULL, FOREIGN KEY REFERENCES users(user_id)	ID of the user who placed the order
order_date	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	Date and time when the order was placed
total_amount	DECIMAL(10, 2)	NOT NULL, CHECK (total_amount >= 0)	Total amount of the order
status	ENUM('pending', 'processing', 'shipped', 'delivered', 'cancelled')	NOT NULL, DEFAULT 'pending'	Current status of the order
shipping_address	TEXT	NOT NULL	Shipping address for the order

6. `order_items` Table

Links products to specific orders and stores the quantity and price at the time of purchase.

Column Name	Data Type	Constraints	Description

`order_item_id`	`INT`	`PRIMARY KEY, AUTO_INCREMENT`	Unique identifier for each item in an order
`order_id`	`INT`	`NOT NULL, FOREIGN KEY REFERENCES orders(order_id)`	ID of the order this item belongs to
`product_id`	`INT`	`NOT NULL, FOREIGN KEY REFERENCES products(product_id)`	ID of the product ordered
`quantity`	`INT`	`NOT NULL, CHECK (quantity > 0)`	Quantity of the product ordered
`price_at_purchase`	`DECIMAL(10, 2)`	`NOT NULL, CHECK (price_at_purchase >= 0)`	Price of the product at the time of purchase

Relationships

- **`users`** to **`orders`**
  - One-to-Many (One user can place many orders).
- **`categories`** to **`products`**
  - One-to-Many (One category can have many products).
- **`products`** to **`inventory`**:
  - One-to-One (Each product has one inventory record).
- **`products`** to **`order_items`**
  - One-to-Many (One product can appear in many order items across different orders).
- **`orders`** to **`order_items`**
  - One-to-Many (One order can contain many order items).

SQL DDL

```SQL
CREATE DATABASE IF NOT EXISTS technest_db;
```

```sql
USE technest_db;

CREATE TABLE IF NOT EXISTS users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(255) NOT NULL UNIQUE,
    email VARCHAR(255) NOT NULL UNIQUE,
    password_hash VARCHAR(255) NOT NULL,
    first_name VARCHAR(255),
    last_name VARCHAR(255),
    address TEXT,
    phone_number VARCHAR(20),
    role ENUM('customer', 'admin') NOT NULL DEFAULT 'customer',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS categories (
    category_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL UNIQUE,
    description TEXT
);

CREATE TABLE IF NOT EXISTS products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10, 2) NOT NULL CHECK (price >= 0),
    category_id INT NOT NULL,
    image_url VARCHAR(255),
    brand VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);

CREATE TABLE IF NOT EXISTS inventory (
    inventory_id INT AUTO_INCREMENT PRIMARY KEY,
    product_id INT NOT NULL UNIQUE,
    stock_quantity INT NOT NULL DEFAULT 0 CHECK (stock_quantity >= 0),
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

```sql
CREATE TABLE IF NOT EXISTS orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10, 2) NOT NULL CHECK (total_amount >= 0),
    status ENUM('pending', 'processing', 'shipped', 'delivered', 'cancelled')
NOT NULL DEFAULT 'pending',
    shipping_address TEXT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);

CREATE TABLE IF NOT EXISTS order_items (
    order_item_id INT AUTO_INCREMENT PRIMARY KEY,
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL CHECK (quantity > 0),
    price_at_purchase DECIMAL(10, 2) NOT NULL CHECK (price_at_purchase >= 0),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Triggers and Stored Procedures

This document provides a comprehensive overview of the database triggers and stored procedures used in the TechNest project.

Triggers

Triggers are automated database operations that are executed in response to specific events on a particular table. They are crucial for maintaining data integrity, enforcing business rules, and creating audit trails.

1. `before_insert_order_item_check_stock`

- **Event:** `BEFORE INSERT` on `order_items`

- **Purpose:** Prevents an `order_item` from being inserted if there is not enough stock available for the product. This ensures that customers cannot order more items than are available in the inventory.

```
SQL
DELIMITER $$
CREATE TRIGGER before_insert_order_item_check_stock
BEFORE INSERT ON order_items
FOR EACH ROW
BEGIN
    DECLARE available_stock INT;
    SELECT stock_quantity INTO available_stock FROM inventory WHERE product_id
= NEW.product_id;
    IF NEW.quantity > available_stock THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Insufficient stock for this
product.';
    END IF;
END$$
DELIMITER ;
```

2. `after_insert_order_item_decrease_stock`

- **Event:** `AFTER INSERT` on `order_items`
- **Purpose:** Automatically decreases the `stock_quantity` in the `inventory` table after a new `order_item` is successfully added. This keeps the inventory levels accurate in real-time.

```
SQL
DELIMITER $$
CREATE TRIGGER after_insert_order_item_decrease_stock
AFTER INSERT ON order_items
FOR EACH ROW
BEGIN
    UPDATE inventory SET stock_quantity = stock_quantity - NEW.quantity,
last_updated = CURRENT_TIMESTAMP WHERE product_id = NEW.product_id;
END$$
DELIMITER ;
```

3. `after_update_order_status_increase_stock`

- **Event:** `AFTER UPDATE` on `orders`

- **Purpose:** If an order's status is updated to `'cancelled'`, this trigger restocks the inventory by adding the quantities of the cancelled items back. This is essential for managing returns and cancellations.

```
SQL
DELIMITER $$
CREATE TRIGGER after_update_order_status_increase_stock
AFTER UPDATE ON orders
FOR EACH ROW
BEGIN
    IF NEW.status = 'cancelled' AND OLD.status != 'cancelled' THEN
        UPDATE inventory i
        JOIN order_items oi ON i.product_id = oi.product_id
        SET i.stock_quantity = i.stock_quantity + oi.quantity, i.last_updated =
CURRENT_TIMESTAMP
        WHERE oi.order_id = NEW.order_id;
    END IF;
END$$
DELIMITER ;
```

4. `after_insert_order_item_update_order_total`

- **Event:** `AFTER INSERT` on `order_items`
- **Purpose:** Ensures that the `total_amount` for an order in the `orders` table is correctly updated whenever a new `order_item` is added.

```
SQL
DELIMITER $$
CREATE TRIGGER after_insert_order_item_update_order_total
AFTER INSERT ON order_items
FOR EACH ROW
BEGIN
    UPDATE orders
    SET total_amount = (SELECT SUM(quantity * price_at_purchase) FROM
order_items WHERE order_id = NEW.order_id)
    WHERE order_id = NEW.order_id;
END$$
DELIMITER ;
```

5. `after_delete_order_item_update_order_total`

- **Event:** `AFTER DELETE` on `order_items`

- **Purpose:** Ensures the `total_amount` for an order is updated whenever an `order_item` is removed. This is crucial for order modifications.

```
SQL
DELIMITER $$
CREATE TRIGGER after_delete_order_item_update_order_total
AFTER DELETE ON order_items
FOR EACH ROW
BEGIN
    UPDATE orders
    SET total_amount = COALESCE((SELECT SUM(quantity * price_at_purchase) FROM
order_items WHERE order_id = OLD.order_id), 0.00)
    WHERE order_id = OLD.order_id;
END$$
DELIMITER ;
```

6. `before_update_product_price_check_zero`

- **Event:** `BEFORE UPDATE` on `products`
- **Purpose:** Prevents the price of a product from being updated to a negative value, maintaining data integrity.

```
SQL
DELIMITER $$
CREATE TRIGGER before_update_product_price_check_zero
BEFORE UPDATE ON products
FOR EACH ROW
BEGIN
    IF NEW.price < 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Product price cannot be
negative.';
    END IF;
END$$
DELIMITER ;
```

7. `before_delete_user_check_orders`

- **Event:** `BEFORE DELETE` on `users`
- **Purpose:** Prevents a user from being deleted if they have existing orders, which maintains referential integrity and prevents orphaned order records.

```SQL
DELIMITER $$
CREATE TRIGGER before_delete_user_check_orders
BEFORE DELETE ON users
FOR EACH ROW
BEGIN
    DECLARE order_count INT;
    SELECT COUNT(*) INTO order_count FROM orders WHERE user_id = OLD.user_id;
    IF order_count > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cannot delete user;
existing orders are associated with this user.';
    END IF;
END$$
DELIMITER ;
```

8. `log_product_price_update`

- **Event:** `AFTER UPDATE` on `products`
- **Purpose:** Logs any changes to a product's price in the `audit_logs` table. This is useful for tracking price history and for auditing purposes.

```SQL
DELIMITER $$
CREATE TRIGGER log_product_price_update
AFTER UPDATE ON products
FOR EACH ROW
BEGIN
    IF OLD.price <> NEW.price THEN
        INSERT INTO audit_logs (action_type, table_name, record_id, old_value,
new_value)
        VALUES ('UPDATE_PRODUCT_PRICE', 'products', NEW.product_id, OLD.price,
NEW.price);
    END IF;
END$$
DELIMITER ;
```

Stored Procedures

Stored procedures are pre-compiled SQL statements that can be saved and reused. They are used to encapsulate complex logic, improve performance, and enhance security.

1. `GetProductDetails(IN product_id_param INT)`

- **Purpose:** Retrieves all details for a specific product, including its category name and current stock quantity. This provides a comprehensive view of a single product for display on a product page.
- **Parameters:**
    - `product_id_param INT`: The ID of the product to retrieve.

```
SQL
DELIMITER $$
CREATE PROCEDURE GetProductDetails(IN product_id_param INT)
BEGIN
    SELECT
        p.product_id, p.name AS product_name, p.description AS
product_description, p.price,
        c.name AS category_name, p.image_url, p.brand, i.stock_quantity,
p.created_at, p.updated_at
    FROM products p
    JOIN categories c ON p.category_id = c.category_id
    LEFT JOIN inventory i ON p.product_id = i.product_id
    WHERE p.product_id = product_id_param;
END$$
DELIMITER ;
```

2. `GetUsersWithPendingOrders()`

- **Purpose:** Returns a list of users who currently have orders with a `'pending'` or `'processing'` status. This is useful for administrative purposes, such as identifying orders that need to be fulfilled.

```
SQL
DELIMITER $$
CREATE PROCEDURE GetUsersWithPendingOrders()
BEGIN
    SELECT DISTINCT u.user_id, u.username, u.email, u.first_name, u.last_name,
u.phone_number
    FROM users u
    JOIN orders o ON u.user_id = o.user_id
    WHERE o.status IN ('pending', 'processing');
END$$
DELIMITER ;
```

3. `CreateOrderFromCart(IN p_user_id INT, IN p_shipping_address TEXT)`

- **Purpose:** Creates a new order for a user based on the items in their cart. It calculates the total price, moves cart items to order items, and then clears the user's cart.
- **Parameters:**
  - `p_user_id INT`: The ID of the user placing the order.
  - `p_shipping_address TEXT`: The shipping address for the order.

```SQL
DELIMITER $$
CREATE PROCEDURE CreateOrderFromCart(IN p_user_id INT, IN p_shipping_address
TEXT)
BEGIN
    DECLARE v_cart_id INT;
    DECLARE v_order_id INT;

    SELECT cart_id INTO v_cart_id FROM cart WHERE user_id = p_user_id;

    IF v_cart_id IS NOT NULL AND (SELECT COUNT(*) FROM cart_items WHERE cart_id
= v_cart_id) > 0 THEN
        INSERT INTO orders (user_id, shipping_address, total_amount, status)
        VALUES (p_user_id, p_shipping_address, 0.00, 'pending');
        SET v_order_id = LAST_INSERT_ID();

        INSERT INTO order_items (order_id, product_id, quantity,
price_at_purchase)
        SELECT v_order_id, ci.product_id, ci.quantity, p.price
        FROM cart_items ci
        JOIN products p ON ci.product_id = p.product_id
        WHERE ci.cart_id = v_cart_id;

        DELETE FROM cart_items WHERE cart_id = v_cart_id;

        SELECT v_order_id AS new_order_id;
    ELSE
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Cart is empty or does not
exist.';
    END IF;
END$$
DELIMITER ;
```