



EAST WEST UNIVERSITY

Department of Computer Science & Engineering

Project Title: Online Shopping Management System

CSE302: Database System

Section : 08

Semester Year: Fall2024

Submitted By:

Student ID	Student Name	Contribution Percentage
Kawsar Ahmad	2022-3-60-242	30%
Md.Romjan Ali	2022-3-60-247	25%
Raqibul Hasan	2022-3-60-261	25%
Jubair Ahmed Dip	2022-3-60-307	20%

Submitted to:

Khairum Islam

Lecturer, (Department of computer Science and Engineering)

Introduction:

The Online Shopping Management System is a database project aimed at optimizing and organizing the processes associated with an e-commerce platform. The system efficiently organizes data related to users, products, orders, payments, and product categories. The primary focus of this system is to handle key operations, such as user account management, product cataloging, order processing, and payment tracking. The User entity maintains customer information, including names, email addresses, passwords, and contact details, ensuring secure and personalized access. The Product and Product Category entities store comprehensive product details such as names, descriptions, pricing, and stock levels, categorized for efficient browsing. The Order entity links users to the products they purchase, capturing order dates, quantities, total amounts, and statuses. Furthermore, the Order Item entity tracks the specific details of individual items in an order, including price and quantity. The Payment entity records transaction details such as payment method, date, and status, ensuring financial transparency. By implementing this structured database system, the project enables seamless interaction between customers and the platform while ensuring data integrity and consistency. Designed for scalability and efficiency, this system forms the backbone of any e-commerce platform, simplifying complex operations and improving the overall shopping experience.

Requirements Analysis:

1. User Management

Users can register with the system by providing their details:

- Name
- Email (unique)
- Password

- Address
- Phone

2. Product Management

- Products must be categorized under predefined categories.
- Each product contains the following details:
 - Name
 - Description
 - Price
 - Stock
 - Associated Category

3. Product Category Management

- The system supports managing product categories:
 - Category Name
 - Description

4. Order Management

- Users can place orders for one or more products.
- Orders will include:
 - Order date
 - Total amount
 - Status (e.g., Pending, Confirmed, Delivered)
- Each order will be associated with:
 - A specific user
 - One or more products (with quantity and price per product)

5. Order Item Details

- Each order includes:
 - Product details
 - Quantity
 - Price (per item)

6. Payment Management

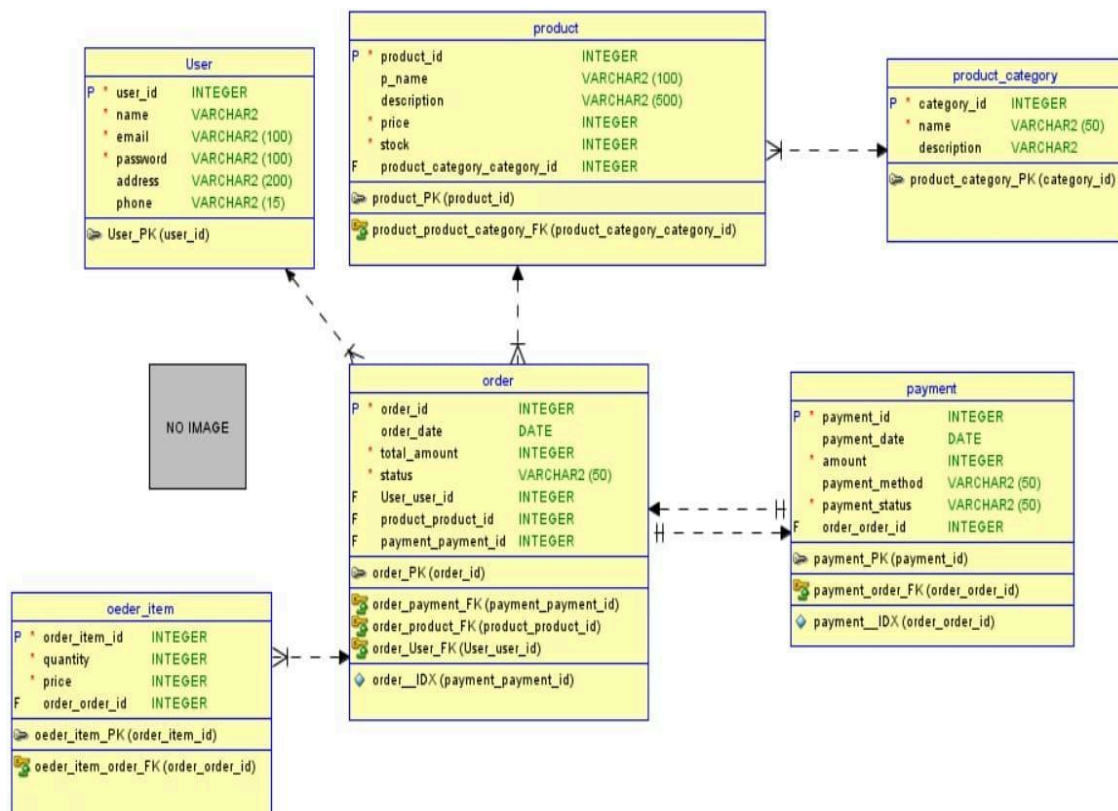
- Payments for orders will include:
 - Payment date

- Amount
- Payment method (e.g., Credit Card, PayPal, etc.)
- Payment status (e.g., Pending, Completed, Failed)
- Payments are linked to specific orders.

Key Functionalities:

- User registration, login, and profile management.
- Product catalog with categories.
- Shopping cart and order placement.
- Order tracking (status updates).
- Payment processing and history.

E-R Mode of the Online shopping management system :



The schema for each entity and relationship:

1. User Table

- **Attributes:**
 - **user_id** (INTEGER, Primary Key): Unique identifier for each user.
 - **name** (VARCHAR2): Stores the user's name.
 - **email** (VARCHAR2(100)): Stores the user's email address.
 - **password** (VARCHAR2(100)): Stores the user's password.
 - **address** (VARCHAR2(200)): Stores the user's address.
 - **phone** (VARCHAR2(15)): Stores the user's phone number.
 - **Relationships:**
 - **User ↔ Order:**
 - **user_id** acts as a foreign key in the **Order** table to associate orders with users.
-

2. Product Table

- **Attributes:**
 - **product_id** (INTEGER, Primary Key): Unique identifier for each product.
 - **p_name** (VARCHAR2(100)): Name of the product.
 - **description** (VARCHAR2(500)): Description of the product.
 - **price** (INTEGER): Price of the product.
 - **stock** (INTEGER): Available stock of the product.
 - **product_category_category_id** (INTEGER, Foreign Key): Links the product to its category.
- **Relationships:**
 - **Product ↔ Product Category:**
 - **product_category_category_id** links a product to its category.
 - **Product ↔ Order Item:**
 - The **product_id** in **Product** is referenced in the **Order Item** table to associate products with specific order items.

3. Product Category Table

- **Attributes:**

- **category_id** (INTEGER, Primary Key): Unique identifier for each category.
- **name** (VARCHAR2(50)): Name of the category.
- **description** (VARCHAR2): Description of the category.

- **Relationships:**

- **Product Category ↔ Product:**
 - **category_id** is referenced as **product_category_category_id** in the **Product** table.

4. Order Table

- **Attributes:**

- **order_id** (INTEGER, Primary Key): Unique identifier for each order.
- **order_date** (DATE): Date when the order was placed.
- **total_amount** (INTEGER): Total amount for the order.
- **status** (VARCHAR2(50)): Status of the order (e.g., pending, completed).
- **User_user_id** (INTEGER, Foreign Key): Links the order to the user who placed it.
- **product_product_id** (INTEGER, Foreign Key): References the product involved in the order.
- **payment_payment_id** (INTEGER, Foreign Key): References the payment for the order.

- **Relationships:**

- **Order ↔ User:**
 - **User_user_id** links an order to the user who placed it.
- **Order ↔ Payment:**
 - **payment_payment_id** links an order to its payment details.

- **Order ↔ Order Item:**
 - `order_id` is referenced in the **Order Item** table to list the items in an order.
-

5. Order Item Table

- **Attributes:**
 - `order_item_id` (INTEGER, Primary Key): Unique identifier for each order item.
 - `quantity` (INTEGER): Quantity of the product ordered.
 - `price` (INTEGER): Price of the product ordered.
 - `order_order_id` (INTEGER, Foreign Key): Links the order item to its parent order.
 - **Relationships:**
 - **Order Item ↔ Order:**
 - `order_order_id` links an order item to its parent order.
 - **Order Item ↔ Product:**
 - `product_id` in the **Product** table is referenced indirectly through the order.
-

6. Payment Table

- **Attributes:**
 - `payment_id` (INTEGER, Primary Key): Unique identifier for each payment.
 - `payment_date` (DATE): Date of payment.
 - `amount` (INTEGER): Amount paid.
 - `payment_method` (VARCHAR2(50)): Method of payment (e.g., credit card, PayPal).
 - `payment_status` (VARCHAR2(50)): Status of payment (e.g., pending, completed).
 - `order_order_id` (INTEGER, Foreign Key): Links the payment to its corresponding order.
- **Relationships:**

- **Payment ↔ Order:**
 - `order_order_id` links a payment to the corresponding order.
-

Relationship Summary

1. **User ↔ Order:**
 - A user can place multiple orders, and each order belongs to one user.
2. **Order ↔ Payment:**
 - Each order has one payment, and each payment is linked to one order.
3. **Order ↔ Order Item:**
 - Each order can include multiple items, and each item is associated with one order.
4. **Order Item ↔ Product:**
 - Each order item is linked to a product.
5. **Product ↔ Product Category:**
 - Each product belongs to one category, and each category can include multiple products.

Implementation Process:(Admin Panel)

User :

```
from django.db import models
class User(models.Model):
    user_id = models.BigAutoField(primary_key=True) #auto-incrementing integer
    name = models.CharField(max_length=100)
    email = models.EmailField(max_length=100, blank=True, null=True)
    password = models.CharField(max_length=100)
    address = models.CharField(max_length=200, blank=True, null=True)
    phone = models.CharField(max_length=15, blank=True, null=True)

    def __str__(self):
        return self.name # Display the user's name in the admin and Django shell
```


The provided Django model defines a **User** table with the following fields:

1. **user_id**: Auto-incrementing primary key.
2. **name**: String (max 100 characters) for the user's name.
3. **email**: Email field (optional, max 100 characters).
4. **password**: String (max 100 characters) for the user's password.
5. **address**: String (optional, max 200 characters).
6. **phone**: String (optional, max 15 characters).

The `__str__` method returns the user's name as the string representation of the object.

Product

```
class Product(models.Model):
    product_id = models.BigAutoField(primary_key=True)
    p_name = models.CharField(max_length=100, blank=True, null=True)
    product_picture = models.ImageField(upload_to='ProductImages', blank=True, null=True)

    description = models.TextField(max_length=500, blank=True, null=True)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    stock = models.PositiveIntegerField()
    product_category_category_id = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return self.p_name or "Unnamed Product"
```

The provided Django model defines a **Product** table with the following fields:

1. **product_id**: Auto-incrementing primary key.
2. **p_name**: Product name (string, optional, max 100 characters).
3. **product_picture**: Image field to store product images, uploaded to the **ProductImages** folder (optional).
4. **description**: Product description (text, optional, max 500 characters).
5. **price**: Decimal field for the product price, with up to 10 digits and 2 decimal places.
6. **stock**: Positive integer field to track stock quantity.

7. **product_category_category_id**: Integer field for category association (optional).

The `__str__` method returns the product name if available, otherwise displays "Unnamed Product."

Product Category:

```
# for product category table
class ProductCategory(models.Model):
    category_id = models.BigAutoField(primary_key=True)
    name = models.CharField(max_length=50)
    description = models.TextField(max_length=500, blank=True, null=True)

    def __str__(self):
        return self.name
```

The provided Django model defines a **ProductCategory** table with the following fields:

1. **category_id**: Auto-incrementing primary key.
2. **name**: String field (max 50 characters) for the category name.
3. **description**: Text field (optional, max 500 characters) for the category description.

The `__str__` method returns the category name as the string representation of the object. This model is intended to organize products into categories.

Order:

```
# Order table
class Order(models.Model):
    order_id = models.BigAutoField(primary_key=True)
    order_date = models.DateField(blank=True, null=True)
    total_amount = models.DecimalField(max_digits=10, decimal_places=2)
    status = models.CharField(max_length=50)
    user_user_id = models.IntegerField(blank=True, null=True)
    product_product_id = models.IntegerField(blank=True, null=True)
    payment_payment_id = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return f"Order #{self.order_id} - Status: {self.status}"
```

The **Order** Django model defines a table for managing orders with the following fields:

1. **order_id**: Auto-incrementing primary key.
2. **order_date**: Date field for the order date (optional).
3. **total_amount**: Decimal field for the total amount of the order, with up to 10 digits and 2 decimal places.
4. **status**: String field (max 50 characters) to represent the order status (e.g., "Pending," "Shipped").
5. **user_user_id**: Integer field for associating the order with a user (optional).
6. **product_product_id**: Integer field for associating the order with a product (optional).
7. **payment_payment_id**: Integer field for associating the order with a payment record (optional).

The **__str__** method provides a readable representation of the order, showing the order ID and its status.

Order Item:

```
class OrderItem(models.Model):
    order_item_id = models.BigAutoField(primary_key=True)
    quantity = models.PositiveIntegerField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    order_order_id = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return f"Order Item #{self.order_item_id} - Quantity: {self.quantity}"
```

The `OrderItem` Django model defines a table for managing items within an order, with the following fields:

1. **order_item_id**: Auto-incrementing primary key.
2. **quantity**: Positive integer field representing the quantity of the item in the order.
3. **price**: Decimal field representing the price of the item, with up to 10 digits and 2 decimal places.
4. **order_order_id**: Integer field for associating the order item with a specific order (optional).

The `__str__` method provides a readable representation of the order item, displaying its ID and quantity.

Payment:

```
# Payment table
class Payment(models.Model):
    payment_id = models.BigAutoField(primary_key=True)
    payment_date = models.DateField(blank=True, null=True)
    amount = models.DecimalField(max_digits=10, decimal_places=2)
    payment_method = models.CharField(max_length=50, blank=True, null=True)
    payment_status = models.CharField(max_length=50)
    order_order_id = models.IntegerField(blank=True, null=True)

    def __str__(self):
        return f"Payment #{self.payment_id} - Status: {self.payment_status}"
```

payment_id:

- Type: **BigAutoField**
- Purpose: Primary key, automatically increments to uniquely identify each payment.

payment_date:

- Type: **DateField**
- Purpose: Stores the date of the payment.
- Accepts null values (**blank=True, null=True**).

amount:

- Type: **DecimalField**
- Purpose: Represents the payment amount.
- Precision: Up to 10 digits, with 2 decimal places.

payment_method:

- Type: **CharField**
- Purpose: Specifies the method of payment (e.g., credit card, PayPal).
- Optional field (**blank=True, null=True**).
- Maximum length: 50 characters.

payment_status:

- Type: **CharField**

- Purpose: Indicates the status of the payment (e.g., "Pending," "Completed").
- Maximum length: 50 characters.
- Required field (does not allow null or blank values).

order_order_id:

- Type: **IntegerField**
- Purpose: Represents the foreign key linking the payment to a specific order.
- Optional field (**blank=True, null=True**).

Admin Dashboard (Admin Panel Frontend)

(OSMS)-Online Shopping Management System
WELCOME: **KAWSER** VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

AUTHENTICATION AND AUTHORIZATION
Groups Add Change
Users Add Change

ONAPP
Order items Add Change
Orders Add Change
Payments Add Change
Product categorys Add Change
Products Add Change
Users Add Change

Recent actions

My actions
digital desktop Product
laptop one Product
Digital Smart Watch Product
LED monitor Product
I Phone 16 Product
Xsive smart watch Product
MSI laptop Product
Smart Phone Product
laptop one Product
digital desktop Product

Authentication:

(OSMS)-Online Shopping Management System
WELCOME: **KAWSER** VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Authentication and Authorization > Users

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION
Groups Add
Users Add

ONAPP
Order items Add
Orders Add
Payments Add
Product categorys Add
Products Add
Users Add

Select user to change

Search

Action: 0 of 3 selected

<input type="checkbox"/>	USERNAME	EMAIL ADDRESS	FIRST NAME	LAST NAME	STAFF STATUS
<input type="checkbox"/>	Rakib	-	-	-	✖
<input type="checkbox"/>	kawser	-	-	-	✔
<input type="checkbox"/>	rahat	-	-	-	✖

3 users

FILTER
Show counts
By staff status
All Yes No
By superuser status
All Yes No
By active
All Yes No

Add User or Add Admin Access:

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Authentication and Authorization > Users > Add user

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categories + Add

Products + Add

Users + Add

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:

Required, 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password-based authentication: ☒ Enabled ☐ Disabled

Whether the user will be able to authenticate using a password or not. If disabled, they may still be able to authenticate using other backends, such as Single Sign-On or LDAP.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

SAVE

Save and add another

Save and continue editing

Product Add In Admin Dashboard:

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Onapp > Products

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categories + Add

Products + Add

Users + Add

The product "Digital Smart Watch" was added successfully.

Select product to change

ADD PRODUCT +

Action: Go 0 of 9 selected

☐ PRODUCT

☐ Digital Smart Watch

☐ MSI laptop

☐ Xssive smart watch

☐ I Phone 16

☐ LED monitor

☐ Digital Smart Watch

☐ Smart Phone

☐ laptop one

☐ digital desktop

9 products

Product Add Form:

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Onapp > Products > Add product

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categories + Add

Products + Add

Users + Add

Add product

P name:

Product picture: No file chosen

Description:

Price:

Stock:

Product category category id:

SAVE

Save and add another

Save and continue editing

Product Category Add In Admin Dashboard:

(OSMS)-Online Shopping Management System

WELCOME: **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home » Onapp » Product categorys

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categorys + Add

Products + Add

Users + Add

Select product category to change

ADD PRODUCT CATEGORY +

Action: ----- Go 0 of 4 selected

☐ PRODUCT CATEGORY

☐ Desktop

☐ Laptop

☐ Phone

☐ Smart Watch

4 product categories

Product Category Add Form:

(OSMS)-Online Shopping Management System

WELCOME: **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home » Onapp » Product categorys » Add product category

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categorys + Add

Products + Add

Users + Add

Add product category

Name:

Description:

SAVE

Save and add another

Save and continue editing

Order In Admin Dashboard:

(OSMS)-Online Shopping Management System

WELCOME: **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home » Onapp » Orders

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categorys + Add

Products + Add

Users + Add

Select order to change

ADD ORDER +

Action: ----- Go 0 of 2 selected

☐ ORDER

☐ Order #2 - Status: Paid

☐ Order #1 - Status: delivered

2 orders

Order Item:

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Onapp > Order items

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categorys + Add

Products + Add

Users + Add

Select order item to change

ADD ORDER ITEM +

Action:

 Go 0 of 1 selected

☐ ORDER ITEM

☐ Order Item #1 - Quantity: 12

1 order item

Payment:

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Onapp > Payments

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add

Payments + Add

Product categorys + Add

Products + Add

Users + Add

Select payment to change

ADD PAYMENT +

Action:

 Go 0 of 2 selected

☐ PAYMENT

☐ Payment #2 - Status: paid

☐ Payment #1 - Status: paid

2 payments

(OSMS)-Online Shopping Management System

WELCOME, **KAWSER** / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Onapp > Payments > Add payment

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ONAPP

Order items + Add

Orders + Add


Payments + Add

Product categorys + Add

Products + Add

Users + Add

Add payment

Payment date: Today  Note: You are 6 hours ahead of server time.

Amount:

Payment method:

Payment status:

Order order id:

SAVE

Save and add another

Save and continue editing

SQL Implementation Queries:

```
def execute_query(query, params=None, fetch_all=True):  
    with connection.cursor() as cursor:  
        cursor.execute(query, params)  
        if fetch_all:  
            return cursor.fetchall()  
        else:  
            return cursor.fetchone()
```

The `execute_query` function is a utility to execute raw SQL queries in a Django application. Here's a breakdown:

Parameters:

1. **query**:
 - The raw SQL query string to be executed.
2. **params** (optional):
 - A list, tuple, or dictionary of parameters to be substituted into the query for dynamic values.
 - Default: `None`.
3. **fetch_all** (optional):
 - A boolean flag to determine whether all rows (`fetchall()`) or just the first row (`fetchone()`) of the result set should be returned.
 - Default: `True`.

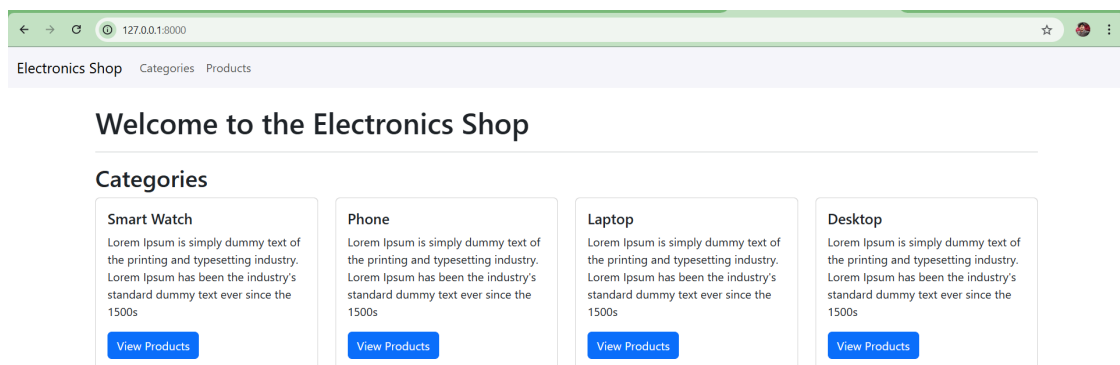
List Category:

```
def list_categories(request):
    # Raw SQL query to fetch categories
    query = "SELECT category_id, name, description FROM onapp_productcategory"
    categories = execute_query(query)

    # Debugging line
    print("Fetched categories:", categories)

    return render(request, 'home.html', {'categories': categories})
```

The `list_categories` function is a Django view designed to fetch product categories from the database and display them on the `home.html` template.



Add Category:

```
def add_category(request):
    if request.method == 'POST':
        form = ProductCategoryForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            description = form.cleaned_data['description']
            query = "INSERT INTO onapp_productcategory (name, description) VALUES (%s, %s)"
            execute_query(query, [name, description])
            return redirect('categories')
    else:
        form = ProductCategoryForm()
    return render(request, 'add_category.html', {'form': form})
```

The `add_category` function is a Django view to handle adding a new product category. Here's a summary:

1. Method Check:

- If the request is POST, it processes form data using the `ProductCategoryForm`.
- If GET, it renders an empty form for user input.

2. Form Validation:

- Checks if the form is valid.
- If valid, retrieves `name` and `description` from the form.

3. Database Insert:

Executes a raw SQL query to insert the new category into the `onapp_productcategory` table:

sql

CopyEdit

```
INSERT INTO onapp_productcategory (name, description)
VALUES (%s, %s)
```

4. Redirection:

- Redirects to the "categories" page upon successful insertion.

5. Template Rendering:

- Renders the `add_category.html` template with the form for GET requests or invalid POST submissions.

Edit Category:

```
def edit_category(request, category_id):
    if request.method == 'POST':
        form = ProductCategoryForm(request.POST)
        if form.is_valid():
            name = form.cleaned_data['name']
            description = form.cleaned_data['description']
            query = "UPDATE onapp_productcategory SET name = %s, description = %s WHERE category_id = %s"
            execute_query(query, [name, description, category_id])
            return redirect('categories')
        else:
            query = "SELECT name, description FROM onapp_productcategory WHERE category_id = %s"
            category = execute_query(query, [category_id], fetch_one=True)
            form = ProductCategoryForm(initial={'name': category[0], 'description': category[1]})
    return render(request, 'edit_category.html', {'form': form, 'category_id': category_id})
```

Delete Category:

```
def delete_category(request, category_id):
    query = "DELETE FROM onapp_productcategory WHERE category_id = %s"
    execute_query(query, [category_id])
    return redirect('categories')
```

Manage Products:

```
def manage_products(request):
    # Query to fetch all products from the database
    query = """
    SELECT p.product_id, p.p_name, p.product_picture, p.price, p.stock, c.name AS category_name
    FROM onapp_product p
    LEFT JOIN onapp_productcategory c ON p.product_category_category_id = c.category_id
    """
    products = execute_query(query)
```

The function `manage_products` is responsible for fetching and managing product data from the database. Here's a summary:

1. Purpose: Retrieves details of all products, including their categories.
2. Query:
 - Selects fields such as `product_id`, `p_name`, `product_picture`, `price`, `stock`, and the associated category name.
 - Joins the `onapp_product` table with the `onapp_productcategory` table using a `LEFT JOIN` to include products even if they don't belong to a category.
3. Result: Executes the query and stores the product data in the `products` variable, which can then be used (e.g., displayed in a view).

This function integrates product details with their category names for management purposes.

List Products:

```
# Pass the products data to the template
return render(request, 'products.html', {'products': products})
def list_products(request):
    query = """
    SELECT p.product_id, p.p_name, p.product_picture, p.description, p.price, p.stock, c.name AS category_name
    FROM onapp_product p
    LEFT JOIN onapp_productcategory c ON p.product_category_id = c.category_id
    """
    products = execute_query(query)
    return render(request, 'products.html', {'products': products})
```

The `list_products` function is a Django view that retrieves a list of products and their associated categories from the database using a raw SQL query. It performs the following:

1. Executes a SQL query to select product details (`product_id`, `p_name`, `product_picture`, `description`, `price`, `stock`) and the corresponding category name (`category_name`) by joining the `onapp_product` table with the `onapp_productcategory` table.
2. Passes the fetched product data to the `products.html` template for rendering.

Add Products:

```
def add_product(request):
    if request.method == 'POST':
        form = ProductForm(request.POST, request.FILES)
        if form.is_valid():
            p_name = form.cleaned_data['p_name']
            product_picture = form.cleaned_data['product_picture']
            description = form.cleaned_data['description']
            price = form.cleaned_data['price']
            stock = form.cleaned_data['stock']
            category_id = form.cleaned_data['product_category_category_id']

            query = """
            INSERT INTO onapp_product (p_name, product_picture, description, price, stock, product_category_category_id)
            VALUES (%s, %s, %s, %s, %s, %s)
            """
            execute_query(query, [p_name, product_picture, description, price, stock, category_id])
            return redirect('products')
        else:
            form = ProductForm()
    return render(request, 'add_product.html', {'form': form})
```

The `add_product` function handles adding a new product to the database:

1. Request Handling:
 - If the request method is `POST`, it processes the form data.
 - If valid, it extracts product details like name, picture, description, price, stock, and category ID.
2. SQL Query:
 - Executes an `INSERT` SQL query to add the product details into the `onapp_product` table.
3. Redirect:
 - After successful insertion, it redirects to the `products` page.
4. Form Rendering:
 - For non-`POST` requests, it renders an empty form in the `add_product.html` template for product creation.

Edit Product:

```

def edit_product(request, product_id):
    if request.method == 'POST':
        form = ProductForm(request.POST, request.FILES)
        if form.is_valid():
            p_name = form.cleaned_data['p_name']
            product_picture = form.cleaned_data['product_picture']
            description = form.cleaned_data['description']
            price = form.cleaned_data['price']
            stock = form.cleaned_data['stock']
            category_id = form.cleaned_data['product_category_category_id']

            query = """
            UPDATE onapp_product
            SET p_name = %s, product_picture = %s, description = %s, price = %s, stock = %s, product_category_category_id = %s
            WHERE product_id = %s
            """
            execute_query(query, [p_name, product_picture, description, price, stock, category_id, product_id])
            return redirect('products')
        else:
            query = """
            SELECT p_name, product_picture, description, price, stock, product_category_category_id
            FROM onapp_product WHERE product_id = %s
            """
            # Use fetch_all=True and extract the first row
            result = execute_query(query, [product_id], fetch_all=True)
            product = result[0] if result else None

            if not product:
                # Handle case where the product doesn't exist
                return redirect('products')

            form = ProductForm(initial={
                'p_name': product[0],
                'product_picture': product[1],
                'description': product[2],
                'price': product[3],
                'stock': product[4],
                'product_category_category_id': product[5]
            })
    return render(request, 'edit_product.html', {'form': form, 'product_id': product_id})

```

HTTP POST Request:

- **Handles Form Submission:**
 - Uses **ProductForm** to validate user input.
 - Extracts product details like **p_name**, **product_picture**, **description**, **price**, **stock**, and **category_id**.
 - Executes an SQL **UPDATE** query to update the product in the database.
 - Redirects to the products page after successful update.

HTTP GET Request:

- **Pre-fills the Form:**
 - Fetches the product details from the database using an SQL **SELECT** query.

- If the product exists, populates the **ProductForm** with the existing product details.
- If the product does not exist, redirects to the products page.

Template Rendering:

- Renders the **edit_product.html** template, passing the form and **product_id**.

Delete Product:

```
def delete_product(request, product_id):  
    query = "DELETE FROM onapp_product WHERE product_id = %s"  
    execute_query(query, [product_id])  
    return redirect('products')
```

The **delete_product** function deletes a product from the database:

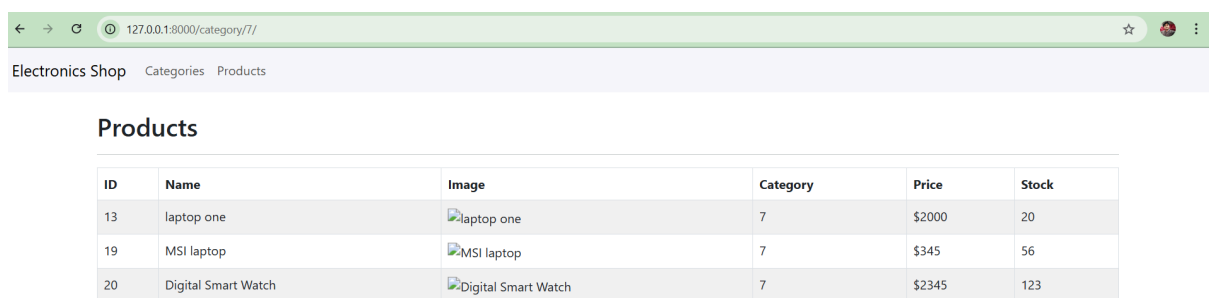
1. **SQL Query:**
 - Executes a **DELETE** query to remove a product from the **onapp_product** table based on the provided **product_id**.
2. **Redirect:**
 - After deletion, it redirects the user to the **products** page.



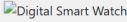
Product Category:

```
def category_products(request, category_id):
    query = """
    SELECT p.product_id, p.p_name, p.product_picture, p.description, p.price, p.stock
    FROM onapp_product p
    WHERE p.product_category_category_id = %s
    """
    products = execute_query(query, [category_id])

    category_query = "SELECT name FROM onapp_productcategory WHERE category_id = %s"
    category_name = execute_query(category_query, [category_id], fetch_one=True)[0]

    return render(request, 'category_products.html', {'products': products, 'category_name': category_name})
```



ID	Name	Image	Category	Price	Stock
13	laptop one		7	\$2000	20
19	MSI laptop		7	\$345	56
20	Digital Smart Watch		7	\$2345	123

The `category_products` function retrieves and displays products belonging to a specific category:

1. Products Retrieval:
 - Executes a SQL query to fetch all products from the `onapp_product` table that match the given `category_id`.
2. Category Name Retrieval:
 - Executes another query to fetch the name of the category based on the `category_id`.
3. Render Response:
 - Passes the retrieved products and category name to the `category_products.html` template for rendering.

Concluding Remarks :

An Online Shopping Management System streamlines the management of products, categories, orders, payments, and customer interactions. It centralizes functionalities like adding, updating, and deleting items while enhancing user experience through quick browsing and purchasing. Well-structured data models ensure secure and efficient handling of relationships, with SQL queries and ORM optimizing storage and retrieval. Its modular design supports future scalability, allowing features like reviews or discounts. The system provides user-friendly interfaces for admins and customers, ensuring seamless navigation. With robust security measures, it protects sensitive data. Overall, it boosts operational efficiency, customer satisfaction, and revenue growth for online stores.