

Shopping Application Methodology and Contribution

23085483 – Tran Duy Anh
23085487 – Nguyen Nhat Minh
23085492 – Nguyen Duc Trung

April 23, 2025

Contents

1	Scrum Implementation	3
1.1	Sprint Organization	3
1.2	Development Process	3
1.2.1	CI/CD Pipeline	3
2	Team Contributions	4
2.1	Backend Development	4
2.1.1	API Architecture	4
2.1.2	Authentication and Authorization	4
2.1.3	Security Measures	5
2.2	Database Design	5
2.3	Data Validation with Schemas	6
2.4	Dummy Data Implementation	7
2.5	Key Functionalities	8
2.5.1	Shop Management	8
2.5.2	Order and Cart Management	9
2.5.3	Admin Dashboard	9
2.5.4	Search Functionality	10
2.6	Frontend Development	10
2.6.1	User Authentication	11
2.6.2	Shop Management	12
2.6.3	Product Management	12
2.6.4	Shop Owner Dashboard	12
2.6.5	Shop Owner Product Management	13
2.6.6	Shop Owner Order Management	13
2.6.7	Admin Dashboard	14
2.6.8	Admin Product Management	15
2.6.9	Admin Shop Owner Management	16
2.6.10	Admin User Management	17
2.6.11	Admin Category Management	18
2.6.12	Admin System Statistics	19
2.6.13	API Integration	20
2.6.14	Non-Functional Requirements	20

3	Technical Implementations	20
3.1	Project Structure	20
3.2	Payment Integration	22
3.3	Testing Implementation	23
3.3.1	Authentication Testing (<code>test_auth.py</code>)	23
3.3.2	Cart Testing (<code>test_cart.py</code>)	24
3.3.3	Payment Testing (<code>test_payment.py</code>)	24
3.3.4	Admin Testing (<code>test_admin.py</code>)	25
3.3.5	Product Testing (<code>test_product.py</code>)	25
3.3.6	Order Testing (<code>test_order.py</code>)	26
3.3.7	Category Testing (<code>test_category.py</code>)	27
3.3.8	Search Testing (<code>test_search.py</code>)	27
3.3.9	Shop Testing (<code>test_shop.py</code>)	28
4	Conclusion	28

Abstract

This document outlines the development process and team feedback for an e-commerce application built using a Python frontend based on CustomTkinter and a FastAPI backend. The Scrum process enabled iterative development, with feedback including backend infrastructure, database schema, security, testing, and a responsive GUI. The frontend provides role-based interfaces for customers, shop owners, and admin users, which interface seamlessly with the backend via RESTful APIs. Comprehensive unit and integration testing ensures system reliability. We introduce the system architecture, core functionalities, testing plan, and implementation, pointing out the frontend's authentication, shop, product, owner, and admin management functionalities.

1 Scrum Implementation

1.1 Sprint Organization

The project utilizes the Scrum method to supply a dynamic and collaborative development process. Being a three-member team, the work is organized into incremental sprints, with each member focusing on a primary area of expertise:

- **Trung:** Responsible for system architecture, supporting development, creating and managing all system diagrams (Use Case, Class, Sequence), and writing unit and integration tests.
- **Minh:** Focused on building the frontend user interface using CustomTkinter, ensuring an intuitive and responsive design.
- **Duy Anh:** Led backend development, handling database modeling, API routing, implementing security and data validation layers, and ensuring high test coverage.

Each sprint concludes with a review and retrospective session, enabling early issue detection and continuous process improvement. The Scrum approach balances flexibility and structure, fostering creativity while maintaining discipline to deliver a scalable application.

1.2 Development Process

1.2.1 CI/CD Pipeline

To maintain a high development pace and ensure confidence in deployments, a CI/CD pipeline was implemented. This pipeline triggers automated tests and deployment scripts with each code push.

```
1 stages:
2   - install
3   - test
4   - deploy
```

Listing 1: CI/CD Stages

The pipeline is configured using GitLab CI/CD. Tests are executed using Pytest, and artifacts are stored for test result reporting.

2 Team Contributions

2.1 Backend Development

2.1.1 API Architecture

A modular backend was developed using FastAPI, incorporating RESTful routing for scalability. The architecture separates business logic through routers, supporting functionalities such as shop management, product handling, order processing, payments, cart operations, search, categories, and administration.

```
1 app.include_router(search.router, prefix="/search", tags=["search"])
2 app.include_router(auth.router, prefix="/auth", tags=["auth"])
3 app.include_router(payment.router, prefix="/payment", tags=["payment"])
4 app.include_router(shop.router, prefix="/shops", tags=["shops"])
5 app.include_router(product.router, prefix="/product", tags=["product"])
6 app.include_router(category.router, prefix="/category", tags=["category"])
7 app.include_router(order.router, prefix="/order", tags=["order"])
8 app.include_router(cart.router, prefix="/cart", tags=["cart"])
9 app.include_router(admin.router, prefix="/admin", tags=["admin"])
```

Listing 2: API Routing Example

Each router is organized by domain responsibility, such as `/shops` for shop management, `/order` for order processing, and `/admin` for administrative functions, ensuring efficient collaboration and ease of scalability.

2.1.2 Authentication and Authorization

A role-based authentication system (customer, shop-owner, admin) was implemented using OAuth2, with password hashing via Bcrypt and JWT access tokens. The `hashing.py` file handles, 2app handles secure password hashing and token generation.

```
1 def create_access_token(data: dict):
2     to_encode = data.copy()
3     expire = datetime.utcnow() + timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
4     to_encode.update({"exp": expire})
5     encoded_jwt = jwt.encode(to_encode, SECRET_KEY, algorithm=ALGORITHM)
6     return encoded_jwt
```

Listing 3: JWT Token Creation

This authentication system integrates with `auth.py` to provide endpoints for registration, login, and user profile management, incorporating security checks like email duplication prevention and password verification.

2.1.3 Security Measures

Advanced security measures were implemented, including encryption of sensitive data such as credit card numbers and CVVs using the Fernet library in `security.py`. Encryption keys are derived from environment variables for enhanced security.

```
1 def encrypt_card_number(card_number: str) -> str:
2     return fernet.encrypt(card_number.encode()).decode()
3
4 def decrypt_card_number(encrypted_card: str) -> str:
5     return fernet.decrypt(encrypted_card.encode()).decode()
```

Listing 4: Card Number Encryption

Data validation was integrated into schemas (`payment.py`) to ensure the validity of card numbers, CVVs, and expiry dates before storage.

```
1 class PaymentCreate(BaseModel):
2     payment_method: str = Field(..., min_length=3, max_length=50)
3     card_number: str = Field(..., min_length=12, max_length=19,
4     pattern=r"^\d+$")
5     cvv: str = Field(..., min_length=3, max_length=3, pattern=r"
6     ^\d+$")
7     expiry_date: str
8
9     @field_validator("card_number")
10    @classmethod
11    def encrypt_card(cls, value):
12        return encrypt_card_number(value)
```

Listing 5: Payment Validation

2.2 Database Design

The database schema was designed using `SQLModel` and MySQL, ensuring robust relationships and referential integrity between major entities: **User**, **Shop**, **Product**, **Order**, **Payment**, **Cart**, and **Category**.

Each entity is mapped to a table with appropriate primary keys, foreign keys, and relationships. The design supports real-world e-commerce workflows such as user authentication, managing shops and products, processing payments and orders, and handling shopping carts.

The main models include:

- **User**: Holds user information (email, username, phone) and is related to Shops, Orders, Payments, and Carts.
- **Shop**: Linked to a User (shop owner) and holds Products.
- **Product**: Belongs to a Shop and optionally a Category; can have multiple Product Images.
- **Order** and **OrderItem**: Track purchase details, tied to User, Shop, Payment, and Products.

- **Payment:** Linked to User; stores payment information securely.
- **Cart and CartItem:** Allow users to add products to cart before checkout.
- **Category:** Classifies products into meaningful groups.

The following Python code snippet demonstrates the definition of the Shop model:

```

1 class Shop(SQLModel, table=True):
2     id: Optional[int] = Field(default=None, primary_key=True)
3     owner_id: int = Field(foreign_key="user.id")
4     name: str = Field(unique=True, index=True)
5     description: Optional[str] = None
6     image_url: Optional[str] = None
7     address: str
8     latitude: float
9     longitude: float
10    created_at: datetime = Field(default_factory=datetime.utcnow)
11
12    owner: User = Relationship(back_populates="shops")
13    products: List["Product"] = Relationship(back_populates="shop
    ")

```

Listing 6: Shop Model

To illustrate the complete database design, the Entity-Relationship Diagram (ERD) is shown below:

Indexes were applied to fields like **email** (User) and **name** (Shop, Category) to optimize query performance, while relationships facilitate efficient access to related data.

2.3 Data Validation with Schemas

Pydantic was utilized in the `schemas/` directory to validate input and output data, ensuring consistency and security. For example, the `UserCreate` schema enforces password length and phone number format.

```

1 class UserCreate(BaseModel):
2     username: str
3     email: EmailStr
4     phone_number: str
5     password: str
6
7     @field_validator("password")
8     @classmethod
9     def password_must_not_be_empty(cls, v):
10        if len(v) < 8:
11            raise ValueError("Password must be at least 8
12        characters long")
13        return v

```

Listing 7: User Creation Schema

Schemas like `PaymentCreate`, `OrderCreate`, and `ProductRead` were designed to map accurately to database models, handling optional fields and sensitive data (e.g., card number encryption).

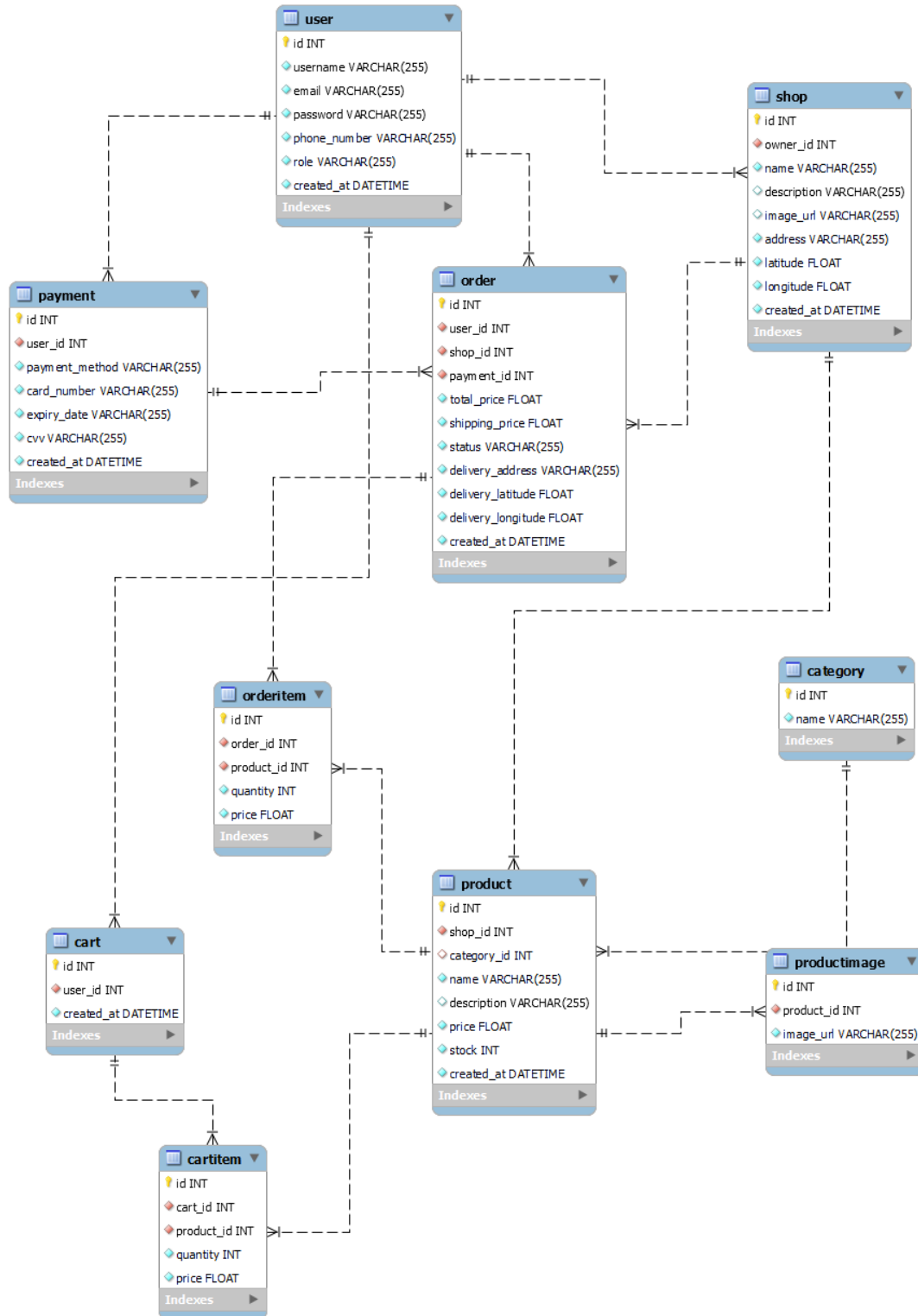


Figure 1: Database

2.4 Dummy Data Implementation

Dummy data was implemented in `dummy_data.py` to support testing and demonstration, including users, shops, products, orders, and carts.

```
1 def insert_dummy_data(session: Session):
```

```

2     if not session.exec(select(User).where(User.email == "
user@example.com")).first():
3         customer = User(
4             username="string",
5             email="user@example.com",
6             password=hash_password("string"),
7             phone_number="1234567890",
8             role="customer",
9         )
10        session.add(customer)
11        session.commit()

```

Listing 8: Dummy Data Loader

Conditional insertion prevents duplicates, ensuring the database is ready for testing and demonstration scenarios.

2.5 Key Functionalities

2.5.1 Shop Management

Endpoints in `shop.py` were developed for shop management, including creation, updates, deletion, and statistics retrieval. The `/shops/create` endpoint integrates geocoding to automatically derive coordinates from addresses and supports image uploads.

```

1 @router.post("/create", response_model=ShopRead)
2 def create_shop(
3     name: str = Form(...),
4     description: str = Form(None),
5     address: str = Form(...),
6     file: UploadFile = File(None),
7     session: Session = Depends(get_session),
8     current_user: User = Depends(get_current_user),
9 ):
10     geolocator = Nominatim(user_agent="shop_locator")
11     location = geolocator.geocode(address)
12     if not location:
13         raise HTTPException(status_code=400, detail="Invalid
address")
14     shop = Shop(
15         name=name,
16         description=description,
17         address=address,
18         latitude=location.latitude,
19         longitude=location.longitude,
20         owner_id=current_user.id,
21     )
22     session.add(shop)
23     session.commit()

```

Listing 9: Shop Creation Endpoint

The `/shops/stats/[shop_id]` endpoint provides revenue and order count insights, aiding shop owners in effective management.

2.5.2 Order and Cart Management

Order and cart processing systems were implemented in `order.py` and `cart.py`. The `/cart/checkout` endpoint groups items by shop, calculates shipping fees based on distance, and creates separate orders.

```
1 @router.post("/checkout")
2 def checkout_cart(
3     checkout_data: dict,
4     session: Session = Depends(get_session),
5     current_user: User = Depends(get_current_user),
6 ):
7     delivery_address = checkout_data.get("delivery_address")
8     payment_id = checkout_data.get("payment_id")
9     selected_items = checkout_data.get("selected_items", [])
10    if not delivery_address or not payment_id:
11        raise HTTPException(
12            status_code=400,
13            detail="Missing required parameters: delivery_address
14            and payment_id",
15        )
16    # ... (logic for grouping items by shop and creating orders)
```

Listing 10: Checkout Endpoint

Shipping fees are calculated based on geographic distance with a capped maximum, ensuring fairness for users.

2.5.3 Admin Dashboard

Administrative endpoints in `admin.py` were developed for user management, shop owner oversight, category management, and system statistics. The `/admin/stats/revenue` endpoint provides detailed revenue insights from products and shipping.

```
1 @router.get("/stats/revenue")
2 def get_revenue_statistics(
3     session: Session = Depends(get_session),
4     current_user: User = Depends(get_current_user),
5 ):
6     verify_admin(current_user)
7     orders = session.exec(select(Order)).all()
8     total_product_revenue = sum(order.total_price * 0.05 for
9     order in orders)
10    total_shipping_revenue = sum(order.shipping_price for order
11    in orders)
12    return {
13        "total_revenue": round(total_product_revenue + total_
14        shipping_revenue, 2),
15        "product_revenue": round(total_product_revenue, 2),
16        "shipping_revenue": round(total_shipping_revenue, 2),
17    }
```

Listing 11: Revenue Statistics Endpoint

Statistics on user growth and top-selling products support data-driven administrative decisions.

2.5.4 Search Functionality

A flexible search system was implemented in `search.py`, enabling searches for shops and products by name or category, with filtering by search type (shops, products, or both).

```
1 @router.get("/", response_model=List[Union[Shop, Product]])
2 def search(
3     name: str = Query(None, description="Name to search"),
4     category: str = Query(None, description="Category to filter
5     by (for products)"),
6     search_type: str = Query("both", description="Search type: '
7     shops', 'products', or 'both'"),
8     db: Session = Depends(get_session),
9 ):
10     results = []
11     if search_type in ["shops", "both"]:
12         shop_query = select(Shop)
13         if name:
14             shop_query = shop_query.where(Shop.name.ilike(f"%{
15             name}%"))
16         results.extend(db.exec(shop_query).all())
17     if search_type in ["products", "both"]:
18         product_query = select(Product)
19         if name:
20             product_query = product_query.where(
21                 or_(Product.name.ilike(f"%{name}%"), Product.
22                 description.ilike(f"%{name}%"))
23             )
24         results.extend(db.exec(product_query).all())
25     return results
```

Listing 12: Search Endpoint

2.6 Frontend Development

The frontend, developed using CustomTkinter, provides a modern, dark-themed user interface tailored to customers, shop owners, and administrators. The `main.py` file orchestrates frame-based navigation, enabling seamless transitions between views while enforcing role-based access control.

```
1 def switch_frame(frame_name, *args):
2     global access_token, user_role
3     print(f"switch_frame called with frame_name={frame_name},
4     args={args}")
5     if frame_name == "dashboard_bypass":
6         token_to_use = args[0] if args and isinstance(args[0],
7         str) else access_token
8         frames["dashboard"] = dashboard_frame(root, switch_frame,
9         API_URL, token_to_use)
```

```

7         frames["dashboard"].place(relx=0, rely=0, relwidth=1,
            relheight=1)
8         frames["dashboard"].tkraise()
9         return
10    # ... (logic for role-based routing and frame management)

```

Listing 13: Frame Switching

The UI adopts a consistent dark theme with a primary color (#00c1ff) for accents, configured as follows:

```

1 ctk.set_appearance_mode("dark")
2 ctk.set_default_color_theme("blue")

```

Listing 14: UI Theme Configuration

2.6.1 User Authentication

User authentication is handled by `login.py` and `register.py` in the `components/auth` directory. The login interface provides fields for email and password, with validation to ensure non-empty inputs. Upon successful login, a JWT token is received and passed to the dashboard.

```

1 def login_api(email, password, API_URL):
2     try:
3         response = requests.post(
4             f"{API_URL}/auth/login", json={"email": email, "
password": password}
5         )
6         if response.status_code == 200:
7             data = response.json()
8             return response.status_code, data
9         return response.status_code, response.json()
10    except Exception as e:
11        return 500, {"detail": f"Request error: {str(e)}"}

```

Listing 15: Login API Request

The registration interface collects username, email, phone number, and password, with client-side validation for email format, password length, and phone number digits. The `forgot_pass.py` file, intended for password recovery, is currently empty and not integrated into the application.

Key features include:

- Two-column layout with branding on the left and form on the right.
- Input validation using regular expressions (e.g., email pattern).
- Feedback via `CTkMessageBox` for errors or success.
- Navigation links to switch between login and registration.

2.6.2 Shop Management

Shop management is implemented in `components/shop/create_shop.py` and `components/shop/view_shop.py`. The shop creation interface allows users to input a name, description, address, and upload a logo, with validation and image preview. The shop view displays details and products in a scrollable grid.

Key features include:

- Form submission to `/shops/create` with JWT authentication.
- Dynamic product cards with images and view buttons.
- Role-based navigation for shop owners.

2.6.3 Product Management

Product management is handled by `components/product/create_product.py`, `view_product.py`, and `edit_product.py`. Shop owners can create and edit products, while customers can view details and add items to their cart.

Key features include:

- Dynamic category selection for product creation.
- Quantity selector for cart addition in product view.
- Image handling with preview and resizing.

2.6.4 Shop Owner Dashboard

The shop owner dashboard, implemented in `components/owner/owner_dashboard.py`, provides a centralized interface for shop management, displaying shop details, revenue statistics, and recent products.

Key features include:

- Shop information section with logo, name, description, and address.
- Revenue analysis with all-time shipped orders and product revenue.
- Scrollable product table showing image, title, name, category, price, and stock.
- Auto-refresh every 30 seconds for statistics.

The product table creation is shown below:

```
1 def create_product_row(product):
2     row = ctk.CTkFrame(product_rows_frame, fg_color="transparent",
3         height=60)
4     row.pack(fill="x", pady=1)
5     img_cell = ctk.CTkFrame(row, fg_color="transparent")
6     img_cell.place(relx=0, rely=0, relwidth=0.15, relheight=1)
7     img_label = ctk.CTkLabel(img_cell, text="No Image")
8     img_label.place(relx=0.5, rely=0.5, anchor="center")
9     if product.get("images") and len(product["images"]) > 0:
10         image_url = product["images"][0].get("image_url")
```

```

10         if image_url:
11             resp = requests.get(image_url)
12             if resp.status_code == 200:
13                 pil_img = Image.open(io.BytesIO(resp.content)).
resize((40, 40))
14                 tk_img = ImageTk.PhotoImage(pil_img)
15                 img_label.configure(image=tk_img, text="")
16                 img_label.image = tk_img

```

Listing 16: Product Table Row Creation

2.6.5 Shop Owner Product Management

The product management interface, in `components/owner/owner_products.py`, allows shop owners to view, edit, delete, or archive products. A scrollable table displays product details, with action buttons for editing and deletion.

Key features include:

- Table columns for image, title, name, category, price, stock, and actions.
- Edit button navigates to `edit_product.py`.
- Delete button prompts confirmation and offers archiving (setting stock to 0) if deletion is blocked by orders.

The delete/archive logic is implemented as follows:

```

1 def delete_product():
2     if CTkMessageBox(title="Confirm Delete", message=f"Delete
product {product['name']}?", icon="question", option_1="Yes",
option_2="No").get() == "Yes":
3         headers = {"Authorization": f"Bearer {token}"}
4         resp = requests.delete(f"{API_URL}/product/delete/{
product['id']}", headers=headers)
5         if resp.status_code == 200:
6             CTkMessageBox(title="Success", message="Product
deleted successfully", icon="info")
7             fetch_products()
8         elif resp.status_code == 400:
9             if CTkMessageBox(title="Cannot Delete Product",
message="This product cannot be deleted because it has been
ordered by customers.\n\nWould you like to archive this
product instead (set stock to 0)?", icon="warning", option_1="
Yes", option_2="No").get() == "Yes":
10                 archive_product(product["id"])

```

Listing 17: Product Deletion/Archiving

2.6.6 Shop Owner Order Management

The order management interface, in `components/owner/owner_orders.py`, enables shop owners to view and update orders, with tabs for pending and shipped orders.

Key features include:

- Tabbed interface for pending and shipped orders.
- Order cards displaying order ID, date, customer, total, and products.
- Confirm shipment button for pending orders, updating status via `/order/update_status/[order_id]`.
- Scrollable frames with mouse wheel support.

The order card creation is shown below:

```

1 def create_order_card(parent, order_data, is_pending=True):
2     card = ctk.CTkFrame(parent, fg_color="#2b2b2b", corner_radius
3         =10, height=180)
4     card.pack(fill="x", pady=10, padx=5)
5     order_id = order_data.get("id", "N/A")
6     order_label = ctk.CTkLabel(card, text=f"Order #{order_id}",
7         font=("Helvetica", 16, "bold"), text_color="white")
8     order_label.pack(anchor="w")
9     if is_pending:
10         def confirm_shipment():
11             if CTkMessageBox(title="Confirm Shipment", message=f"
12 Confirm that order #{order_id} has been shipped?", icon="
13 question", option_1="Yes", option_2="No").get() == "Yes":
14                 update_order_status(order_id, "shipped")
15             confirm_btn = ctk.CTkButton(card, text="Confirm Shipment"
16 , command=confirm_shipment, fg_color="#00c1ff", hover_color="#
17 0096ff", height=30)
18             confirm_btn.pack(side="bottom", fill="x")

```

Listing 18: Order Card Creation

2.6.7 Admin Dashboard

The admin dashboard, implemented in `components/admin/dashboard.py`, serves as the central hub for administrators, providing navigation to specialized management interfaces for products, shop owners, users, categories, and system statistics. It integrates with backend endpoints in `auth.py` to verify admin roles and fetch user profiles.

```

1 def check_admin():
2     headers = {"Authorization": f"Bearer {access_token}"}
3     try:
4         resp = requests.get(f"{API_URL}/auth/role", headers=
5 headers)
6         if resp.status_code == 200:
7             role_data = resp.json()
8             if role_data.get("role") != "admin":
9                 switch_func("dashboard")
10                return False
11            else:
12                switch_func("login")
13                return False
14        except Exception as e:

```

```

14         CTkMessageBox(
15             title="Error",
16             message=f"Failed to connect to server: {e}",
17             icon="cancel",
18         )
19         return False
20     return True

```

Listing 19: Admin Role Verification

Key features include:

- Card-based layout with clickable cards for User Management, Shop Owner Management, Category Management, and System Statistics, each with distinct colors for visual clarity.
- Dynamic welcome message displaying the admin's username fetched from `/auth/profile`.
- Role-based access control, redirecting non-admin users to the main dashboard or login page.
- Responsive design with a gradient header, icon support, and a logout button.
- Error handling via `CTkMessageBox` for server connectivity issues.

2.6.8 Admin Product Management

The product management interface, implemented in `components/admin/product_management.py`, allows administrators to oversee marketplace products, including approving, rejecting, or viewing product details. It integrates with the `/admin/products` endpoint for data retrieval and updates.

```

1 def approve_product():
2     selected_items = product_tree.selection()
3     if not selected_items:
4         CTkMessageBox(
5             title="Warning",
6             message="Please select a product to approve",
7             icon="warning",
8         )
9         return
10    product_id = product_tree.item(selected_items[0])["values"]
11    headers = {"Authorization": f"Bearer {access_token}"}
12    try:
13        response = requests.put(
14            f"{API_URL}/admin/products/{product_id}/approve",
15            headers=headers
16        )
17        if response.status_code == 200:
18            CTkMessageBox(
19                title="Success",
20                message="Product approved successfully",

```

```

20         icon="check",
21     )
22     on_filter_change() # Refresh the list
23 except Exception as e:
24     CtkMessageBox(
25         title="Error", message=f"An error occurred: {str(e)}"
26     , icon="cancel"
27     )

```

Listing 20: Product Approval Function

Key features include:

- A searchable and filterable table displaying product ID, name, category, price, shop owner, status, creation date, and image availability.
- Filters for product status (All, Approved, Pending, Rejected) and sorting options (Newest/Oldest First, Price High/Low).
- Action buttons for approving, rejecting, or viewing product details, with rejection requiring a reason via a dialog.
- Detailed product view dialog showing images, descriptions, and shop information, with approve/reject options for pending products.
- Color-coded status tags (green for approved, brown for pending, red for rejected) for visual clarity.

2.6.9 Admin Shop Owner Management

The shop owner management interface, implemented in `components/admin/shop_owner_management.py`, enables administrators to manage shop owner accounts, including viewing their shops and deleting accounts. It integrates with the `/admin/owners` endpoint.

```

1 def delete_shop_owner():
2     selected_item = owner_tree.selection()
3     if not selected_item:
4         CtkMessageBox(
5             title="Warning",
6             message="Please select a shop owner to delete",
7             icon="warning",
8         )
9     return
10    owner_id = owner_tree.item(selected_item[0])["values"][0]
11    username = owner_tree.item(selected_item[0])["values"][1]
12    confirm = CtkMessageBox(
13        title="Confirm Deletion",
14        message=f"Are you sure you want to delete shop owner '{username}'?\n\nThis will also delete all their shops and products.\n\nThis action cannot be undone!",
15        icon="question",
16        option_1="Cancel",
17        option_2="Delete",
18    )

```



```

19     if confirm.get() == "Delete":
20         headers = {"Authorization": f"Bearer {access_token}"}
21         response = requests.delete(
22             f"{API_URL}/admin/owners/{owner_id}", headers=headers
23         )
24         if response.status_code == 200:
25             CtkMessageBox(
26                 title="Success",
27                 message="Shop owner deleted successfully",
28                 icon="check",
29             )
30             fetch_shop_owners()

```

Listing 21: Shop Owner Deletion

Key features include:

- A table displaying shop owner ID, username, email, phone number, and role, with search functionality by username or email.
- Action buttons for viewing shops, refreshing data, and deleting owners, with confirmation dialogs for deletions.
- A dialog for viewing shops owned by a selected owner, displaying shop name, description, address, and creation date.
- Visual highlighting for owners with shops and search matches.

2.6.10 Admin User Management

The user management interface, implemented in `components/admin/user_management.py`, allows administrators to manage regular user (buyer) accounts, including searching and deleting users. It integrates with the `/admin/users` endpoint.

```

1 def delete_user():
2     selected_item = user_tree.selection()
3     if not selected_item:
4         CtkMessageBox(
5             title="Warning",
6             message="Please select a user to delete",
7             icon="warning",
8         )
9         return
10    user_id = user_tree.item(selected_item[0])["values"][0]
11    username = user_tree.item(selected_item[0])["values"][1]
12    confirm = CtkMessageBox(
13        title="Confirm Deletion",
14        message=f"Are you sure you want to delete user '{username}'?",
15        icon="question",
16        option_1="Cancel",
17        option_2="Delete",
18    )
19    if confirm.get() == "Delete":

```

```

20     headers = {"Authorization": f"Bearer {access_token}"}
21     response = requests.delete(
22         f"{API_URL}/admin/users/{user_id}", headers=headers
23     )
24     if response.status_code == 200:
25         CtkMessageBox(
26             title="Success", message="User deleted
27             successfully", icon="check"
28         )
29         fetch_users()

```

Listing 22: User Deletion

Key features include:

- A table displaying user ID, username, email, phone number, and role, with search functionality by username or email.
- Action buttons for refreshing data and deleting users, with confirmation dialogs.
- Visual highlighting for search matches and alternating row colors for readability.

2.6.11 Admin Category Management

The category management interface, primarily implemented in `components/admin/category_management.py`, enables administrators to create, edit, delete, and view product categories, supporting hierarchical structures with parent categories. It integrates with the `/admin/categories` and `/category` endpoints for data operations.

```

1 def save_category():
2     name = name_entry.get().strip()
3     description = desc_entry.get("1.0", "end-1c").strip()
4     parent = parent_var.get()
5     if not name:
6         CtkMessageBox(
7             title="Warning", message="Category name is
8             required", icon="warning"
9         )
10        return
11    parent_id = None
12    if parent != "None":
13        try:
14            parent_id = int(parent.split("ID: ")[1].split(" ")
15            ) [0])
16        except:
17            pass
18    data = {"name": name, "description": description, "parent
19    _id": parent_id}
20    headers = {"Authorization": f"Bearer {access_token}"}
21    try:
22        response = requests.post(
23            f"{API_URL}/category/create",
24            headers=headers,

```

```

22         json=data
23     )
24     if response.status_code == 201:
25         dialog.destroy()
26         CtkMessageBox(
27             title="Success",
28             message="Category added successfully",
29         )
30         fetch_categories()
31     except Exception as e:
32         CtkMessageBox(
33             title="Error", message=f"An error occurred: {str(
34 e)}", icon="cancel"
35         )

```

Listing 23: Category Creation Dialog

Key features of `category_management.py` include:

- A searchable table displaying category ID, name, description, product count, creation date, and parent category, with dynamic data fetching from `/admin/categories`.
- Action buttons for adding, editing, and deleting categories, with dialogs for creating/editing categories that support name, description, and parent category selection.
- Search functionality to filter categories by name, enhancing usability for large datasets.
- Confirmation dialogs for deletions, with specific error handling for categories linked to products.
- Responsive design with a green-themed header and alternating row colors for readability.

2.6.12 Admin System Statistics

The system statistics interface, implemented in `components/admin/system_statistics.py`, provides administrators with key performance indicators and analytics through interactive charts. It integrates with the `/admin/stats` endpoints.

```

1 def fetch_statistics():
2     headers = {"Authorization": f"Bearer {access_token}"}
3     try:
4         response = requests.get(f"{API_URL}/admin/stats/
5 users", headers=headers)
6         if response.status_code == 200:
7             stats = response.json()
8             user_count = stats.get("total_users", 0)
9             owner_count = stats.get("total_shop_owners",
0 )
10         user_card.configure(text=str(user_count))

```

```

10         owner_card.configure(text=str(owner_count))
11         create_user_growth_chart(user_graph_frame,
stats.get("user_growth", []))
12     except Exception as e:
13         CtkMessageBox(
14             title="Error",
15             message=f"Failed to connect to server: {e}",
16             icon="cancel",
17         )

```

Listing 24: Statistics Fetching

Key features include:

- * Statistic cards displaying total users, shop owners, and revenue, updated dynamically via API calls.
- * Tabbed interface for user growth, revenue, and product sales charts, using Matplotlib for visualization.
- * Scrollable charts with mouse wheel support, showing user growth over time, revenue trends, and top-selling products.
- * Date range filtering (Last 7 Days, 30 Days, 90 Days, All Time) for customized analytics.
- * Visual feedback with color-coded bars and data labels for clarity.

2.6.13 API Integration

Frontend-backend communication is managed via `utils/api_requests.py`, handling HTTP requests with JWT authentication. Error handling ensures user feedback through `CtkMessageBox`.

2.6.14 Non-Functional Requirements

- **Security:** JWT token validation and role-based routing prevent unauthorized access, with immediate redirection for non-admin users in the admin dashboard.
- **Usability:** Consistent dark theme, intuitive card-based navigation in the admin dashboard, and dialog-based interactions in category management enhance user experience.
- **Scalability:** Modular frame-based architecture and reusable components support future feature additions.
- **Performance:** Optimized data fetching with periodic refreshes and efficient table rendering minimize latency, even for large category datasets.
- **Reliability:** Comprehensive error handling with user-friendly messages ensures robust operation during network or server issues.

3 Technical Implementations

3.1 Project Structure

The codebase is organized modularly:

```

1 app/
2     backend/
3         models/
4             models.py
5         routes/
6             shop.py
7             payment.py
8             order.py
9             admin.py
10            cart.py
11            auth.py
12            search.py
13            category.py
14            product.py
15        schemas/
16            category.py
17            product.py
18            order.py
19            payment.py
20            shop.py
21            user.py
22        utils/
23            hashing.py
24            security.py
25        database.py
26        main.py
27        dummy_data.py
28    frontend/
29        components/
30            auth/
31                login.py
32                register.py
33                forgot_pass.py
34            shop/
35                create_shop.py
36                view_shop.py
37            product/
38                create_product.py
39                view_product.py
40                edit_product.py
41            admin/
42                dashboard.py
43                category_management.py
44                user_management.py
45        Tisch
46            shop_owner_management.py
47            product_management.py
48            system_statistics.py
49        owner/
50            owner_dashboard.py

```

```

51         iama
52             owner_products.py
53             owner_orders.py
54         dashboard.py
55         user_details.py
56         user_orders.py
57         user_payments.py
58     utils/
59         api_requests.py
60     main.py
61     core/
62         config.py
63     static/
64     test/
65         test_admin.py
66         test_auth.py
67         test_cart.py
68         test_category.py
69         test_order.py
70         test_payment.py
71         test_product.py
72         test_search.py
73         test_shop.py
74         conftest.py

```

Listing 25: Project Structure

3.2 Payment Integration

The payment system in `payment.py` supports credit card validation, with fields for card number, CVV, and expiry date. Sensitive data is encrypted before storage, accessed via `components/user_payments.py` in the frontend.

```

1  def add_payment_method(api_url, token, payment_data
2  ):
3      if not token:
4          return None, {"detail": "Access token not
5          provided. Please log in."}
6      headers = {"Authorization": f"Bearer {token}"}
7      try:
8          response = requests.post(
9              f"{api_url}/payment/add", headers=
10             headers, json=payment_data
11         )
12         return response.status_code, response.json
13     except requests.exceptions.RequestException as
14     e:
15         return None, {"detail": str(e)}

```

Listing 26: Payment Creation API Call

3.3 Testing Implementation

Comprehensive unit and integration tests were implemented in the `test/` directory using Pytest, ensuring the reliability and correctness of the backend APIs, database interactions, and business logic. The `conftest.py` file configures the test environment with an in-memory SQLite database and FastAPI's `TestClient` for API testing.

```
1 engine = create_engine(  
2     "sqlite:///memory:",  
3     connect_args={"check_same_thread": False},  
4     poolclass=StaticPool,  
5     echo=False,  
6 )  
7  
8 @pytest.fixture(name="db_session", scope="function"  
9 )  
10 def db_session_fixture():  
11     SQLAlchemyModel.metadata.create_all(engine)  
12     with Session(engine) as session:  
13         try:  
14             yield session  
15         finally:  
16             session.close()  
17             SQLAlchemyModel.metadata.drop_all(engine)
```

Listing 27: Test Environment Setup

Tests are organized by functionality, covering authentication, cart operations, payments, administration, products, orders, categories, search, and shops. Each test file uses fixtures to set up test data, ensuring isolation and repeatability.

3.3.1 Authentication Testing (`test_auth.py`)

The `test_auth.py` file tests the authentication system, including user registration, login, profile management, and token validation.

- `test_signup_success`: Verifies successful user registration, checking password hashing and database storage.
- `test_signup_duplicate_email`: Ensures duplicate email registration fails with a 400 status code.
- `test_login_success`: Tests successful login with correct credentials, verifying JWT token generation.
- `test_token_expiration`: Simulates token expiration to ensure 401 responses for expired tokens.
- `test_update_profile`: Confirms profile updates for username, email, and phone number.

```
1 def test_signup_success(client, db_session):  
2     user_data = {  
3         "username": "testuser",  
4         "email": "test@example.com",
```

```

5         "phone_number": "1234567890",
6         "password": "testpassword123",
7     }
8     response = client.post("/auth/signup", json=
user_data)
9     assert response.status_code == 200
10    assert response.json()["message"] == "User
created successfully"
11    user = db_session.exec(select(User).where(User.
email == user_data["email"])).first()
12    assert verify_password("testpassword123", user.
password)

```

Listing 28: Signup Test

3.3.2 Cart Testing (test_cart.py)

The `test_cart.py` file tests cart operations, including adding items, retrieving cart contents, updating quantities, removing items, and checking out.

- `test_add_to_cart`: Verifies adding a product to the cart with valid data.
- `test_get_cart_items`: Ensures cart items are retrieved correctly with expected quantities.
- `test_update_cart_item`: Tests updating cart item quantities.
- `test_remove_from_cart`: Confirms item removal from the cart.
- `test_checkout_cart`: Validates the checkout process, ensuring order creation.
- `test_add_to_cart_invalid_product`: Tests error handling for non-existent products.
- `test_add_to_cart_insufficient_stock`: Ensures stock validation during cart addition.

```

1 def test_add_to_cart(client, test_product, auth_
headers):
2     response = client.post(
3         "/cart/add",
4         headers=auth_headers,
5         json={"product_id": test_product.id, "shop_
id": test_product.shop_id, "quantity": 2},
6     )
7     assert response.status_code == 200
8     assert response.json()["message"] == "Item
added to cart successfully"

```

Listing 29: Add to Cart Test

3.3.3 Payment Testing (test_payment.py)

The `test_payment.py` file tests payment method creation, validation, and encryption.

- `test_payment_create_validation`: Verifies validation for card number, CVV, and expiry date, including error cases.
- `test_payment_read`: Ensures payment data is correctly read with decrypted card numbers.
- `test_expiry_date_validation`: Tests expiry date validation for past and future dates.
- `test_card_encryption`: Confirms card number encryption and decryption.

```

1 def test_payment_create_validation():
2     valid_data = {
3         "payment_method": "Credit Card",
4         "card_number": "4111111111111111",
5         "cvv": "123",
6         "expiry_date": f"{(datetime.now().month):02d}/{str(datetime.now().year + 1)[2:]}",
7     }
8     payment = PaymentCreate
9     *valid_data)assertdecrypt_card_number(payment.card_number) ==
    valid_data["card_number"]

```

Listing 30: Payment Validation Test

3.3.4 Admin Testing (`test_admin.py`)

The `test_admin.py` file tests administrative functionalities, including user management, shop owner management, and statistics.

- `test_get_all_users_as_admin`: Verifies admin access to user lists.
- `test_get_all_users_as_regular_user`: Ensures non-admins are denied access.
- `test_get_shop_owners`: Tests retrieval of shop owner data.
- `test_delete_shop_owner`: Confirms shop owner deletion.
- `test_get_user_statistics`: Validates user growth and count statistics.
- `test_delete_admin_account_fail`: Ensures admins cannot delete their own accounts.

```

1 def test_get_all_users_as_admin(client, admin_token):
2     response = client.get("/admin/users", headers={
3         "Authorization": f"Bearer {admin_token}"})
4     assert response.status_code == 200
5     assert isinstance(response.json(), list)

```

Listing 31: Admin User List Test

3.3.5 Product Testing (`test_product.py`)

The `test_product.py` file tests product schema validation and data handling.

- `test_product_create`: Verifies product creation with required and optional fields.
- `test_product_read_schema`: Ensures correct mapping of product data, including categories.
- `test_product_update_schema`: Tests partial product updates.
- `test_product_image_schemas`: Validates product image creation and reading.

```

1 def test_product_create_schema():
2     product_data = {
3         "shop_id": 1,
4         "category_id": 2,
5         "name": "Test Product",
6         "description": "Test Description",
7         "price": 9.99,
8         "stock": 10,
9     }
10    product = ProductCreate
11    *product_data)assertproduct.name ==
12    "TestProduct"assertproduct.price == 9.99

```

Listing 32: Product Creation Schema Test

3.3.6 Order Testing (`test_order.py`)

The `test_order.py` file tests order creation, retrieval, cart integration, and status updates.

- `test_create_order`: Verifies order creation with payment and delivery details.
- `test_get_user_orders`: Ensures users can retrieve their order history.
- `test_get_shop_orders`: Tests shop-specific order retrieval.
- `test_add_to_cart`: Confirms cart addition for orders.
- `test_cart_checkout`: Validates cart checkout with order generation.
- `test_update_order_status`: Tests order status updates.
- `test_delete_order`: Ensures order deletion.

```

1 def test_create_order(client, test_user, test_shop,
2     test_product, test_payment):
3     login_response = client.post("/auth/login",
4     json={"email": "test@example.com", "password": "
5     password123"})
6     token = login_response.json()["access_token"]
7     order_data = {
8         "shop_id": test_shop.id,
9         "payment_id": test_payment.id,
10        "items": [{"product_id": test_product.id, "
11        quantity": 2}],
12        "delivery_address": "1 Westminster Bridge
13        Rd, London SE1 7PB, UK",
14    }

```

```

10     response = client.post("/order/", json=order_
    data, headers={"Authorization": f"Bearer {token}
    "})
11     assert response.status_code == 200
12     assert response.json()["shop-id"] == test_shop.
    id

```

Listing 33: Order Creation Test

3.3.7 Category Testing (test_category.py)

The `test_category.py` file tests category CRUD operations and access control.

- `test_create_category`: Verifies category creation and duplicate handling.
- `test_create_category_unauthorized`: Ensures non-admins cannot create categories.
- `test_get_category`: Tests category retrieval by ID.
- `test_update_category`: Confirms category updates and duplicate name checks.
- `test_delete_category`: Validates category deletion.
- `test_get_all_categories`: Ensures retrieval of all categories.

```

1 def test_create_category(client, db_session):
2     admin = create_test_admin(db_session)
3     headers = get_auth_headers(admin)
4     response = client.post("/category/create",
    headers=headers, json={"name": "Electronics"})
5     assert response.status_code == 200
6     assert response.json()["name"] == "Electronics"

```

Listing 34: Category Creation Test

3.3.8 Search Testing (test_search.py)

The `test_search.py` file tests the search functionality for shops and products.

- `test_empty_search`: Verifies empty search returns all results.
- `test_search_by_shop_name`: Tests shop name search.
- `test_search_by_product_name`: Ensures product name search functionality.
- `test_search_by_product_description`: Tests searching within product descriptions.
- `test_search_by_category`: Validates category-based product search.
- `test_search_with_nonexistent_name`: Confirms empty results for nonexistent names.
- `test_search_with_case_insensitive`: Tests case-insensitive search.
- `test_search_special_characters`: Ensures special character handling.

```

1 def test_search_by_shop_name(client, db_session):
2     shop = Shop(name="Bakery Shop", owner_id=1,
3               address="Test Address", latitude=0.0, longitude
4               =0.0)
5     db_session.add(shop)
6     db_session.commit()
7     response = client.get("/search/?name=Bakery")
8     assert response.status_code == 200
9     assert response.json()[0]["name"] == "Bakery
10    Shop"

```

Listing 35: Shop Name Search Test

3.3.9 Shop Testing (test_shop.py)

The test_shop.py file tests shop management functionalities.

- test_create_shop_success: Verifies shop creation with geocoding.
- test_get_all_shops: Ensures retrieval of all shops.
- test_get_shop_by_id: Tests shop retrieval by ID.
- test_update_shop: Confirms shop updates with geocoding.
- test_delete_shop: Validates shop deletion.
- test_get_owner_shops: Tests retrieval of shops by owner.
- test_get_shop_stats: Ensures shop statistics retrieval.

```

1 def test_create_shop_success(client, db_session):
2     test_user = User(email="test@example.com",
3                     username="testuser", password="testpass123",
4                     phone_number="1234567890", role="shop_owner")
5     db_session.add(test_user)
6     db_session.commit()
7     access_token = create_access_token({"sub": str(
8     test_user.id)})
9     headers = {"Authorization": f"Bearer {access_
10    token}"}
11     shop_data = {"name": "Test Shop", "description":
12     "A test shop", "address": "1600 Amphitheatre
13     Parkway, Mountain View, CA"}
14     response = client.post("/shops/create", json=
15     shop_data, headers=headers)
16     assert response.status_code == 200
17     assert response.json()["name"] == shop_data["
18     name"]

```

Listing 36: Shop Creation Test

4 Conclusion

Through Scrum methodology and strict role definition, a stable e-commerce platform was developed. The backend provides scalable

APIs, secure authentication, validation of data, and proper testing, while the frontend provides user-friendly, role-based interfaces for authentication, shop, product, owner, and admin management. Admin dashboard and category management interfaces offer superior admin control with responsive layouts and advanced features. System reliability is ensured by exhaustive unit and integration tests. Modular structure, CI/CD procedures, security features, and a good test suite ensure a high-quality, scalable system, meeting the needs of customers, shop owners, and administrators.