**1. Create a User**

python

from django.contrib.auth.models import User

# Create a user

user = User.objects.create\_user(username="Lakshmi", password="securepassword123")

**2. Create a Category**

python

from products.models import Category

# Create a category

category = Category.objects.create(name="Electronics")

**3. Create Products**

python

from products.models import Products

# Create products

product1 = Products.objects.create(name="Laptop", price="1000.00", category=category, description="High-performance laptop")

product2 = Products.objects.create(name="Smartphone", price="3000.00", category=category, description="Latest smartphone model")

product3 = Products.objects.create(name="Refrigerator", price="1000.00", category=category, description="Double-door fridge")

**4. Create an Order**

python

from products.models import Order, OrderProduct

# Create an order for the user

order = Order.objects.create(user=user)

# Add products to the order with quantities

OrderProduct.objects.create(order=order, product=product1, quantity=2) # Laptop \* 2

OrderProduct.objects.create(order=order, product=product2, quantity=1) # Smartphone \* 1

OrderProduct.objects.create(order=order, product=product3, quantity=3) # Refrigerator \* 3

**5. Calculate Total Bill**

To apply discounts and taxes:

python

print("Total bill (10% discount, 5% tax):", order.total\_bill(discount=10, tax=5))

**Expected Calculation**:

* **Subtotal**:
* Laptop (1000 \* 2) = 2000
* Smartphone (3000 \* 1) = 3000
* Refrigerator (1000 \* 3) = 3000
* Total Subtotal = 2000 + 3000 + 3000 = 8000
* **Discount (10%)**:
* 10% of 8000 = 800
* After Discount = 8000 - 800 = 7200
* **Tax (5%)**:
* 5% of 7200 = 360
* Final Total = 7200 + 360 = 7560

**Output**:

Total bill (10% discount, 5% tax): 7560.00

1. We can also retrieve all the Orders we have with particular **USER**
   1. For this get user first
      1. user = User.objects.get(id = 3)
      2. orders = user.orders.all()
      3. for ord in orders: print(ord.id) #will get all the order ids of this user
2. We can also check all the Products ordered by the **USER** in Particular **ORDER**
   1. First get the order id
      1. order = Order.objects.get(id=2)
      2. products = order.products.all()
      3. for prod in products: print(prod.name) #will get all the products names
3. we can also check One Order has how many **Products**
   1. from products.models import OrderProduct #manytomany lookup table
   2. # Count the products that include the order with ID 2
   3. product\_count = OrderProduct.objects.filter(order\_id=2).count()
   4. print(\_"Order ID 2 has ", product\_count, " product\_count.")
4. we can also check One Product is part of how many **orders**
   1. from products.models import OrderProduct #manytomany lookup table
   2. # Count the orders that include the product with ID 11
   3. order\_count = OrderProduct.objects.filter(product\_id=11).count()
   4. print(\_"Product ID 11 is part of", order\_count, "orders.")

**📄 Order Service Development Documentation**

**1. Objective**

Build a full **Order Management API** using Django REST Framework (DRF), allowing:

* Create Orders
* List Orders
* Retrieve Order Details
* Update Orders
* Delete Orders

**2. Steps Followed**

**🔹 a) Model Setup**

We ensured that we had the following models ready inside products/models.py:

* **Order** → Basic order information (user, status, created\_at)
* **OrderProduct** → To connect an Order with multiple Products (with quantity)

We added a missing field order\_status in the Order model and created a migration for it:

bash

CopyEdit

python manage.py makemigrations

python manage.py migrate

**🔹 b) Serializers Setup**

Created serializers in products/serializers.py:

* ProductSerializer — For minimal product details (id, name, price)
* OrderProductSerializer — For showing product + quantity inside an order
* OrderSerializer — For full order data (nested order products)

🔥 We used **nested serializers** to display OrderProducts inside the Order itself.

**🔹 c) Views Setup**

Instead of using generics, we manually created **class-based views** for better learning:

* OrderListCreateView (APIView) → to List all orders and Create new order
* OrderDetailView (APIView) → to Retrieve, Update, and Delete a single order

Inside each view:

* GET — fetch data
* POST — create new
* PUT — update existing
* DELETE — remove

✅ Proper error handling was done (example: if order not found, return 404).

**🔹 d) URLs Setup**

In products/urls.py, mapped:

python

CopyEdit

path('orders/', OrderListCreateView.as\_view(), name='order-list-create')

path('orders/<int:pk>/', OrderDetailView.as\_view(), name='order-detail')

Also, products.urls was included in the main project ecommerce\_application/urls.py under /api/ prefix.

Thus:

* List/Create Orders ➔ api/orders/
* Detail/Update/Delete Order ➔ api/orders/<pk>/

**🔹 e) Data Population**

Since the database was empty, we created a script populate.py:

* 10 random Users
* 10 random Products
* 5 random Orders
* Random Products assigned to Orders

We fixed some migration issues and finally populated the RDS database.

**🔹 f) API Testing**

Using Postman / PyCharm HTTP Client, tested the APIs:

* GET /api/orders/ — List all orders ✅
* POST /api/orders/ — Create new order ✅
  + *POST* http://localhost:8000/api/orders/  
    *Content-Type*: application/json  
      
    {  
     "user": 5  
    }
  + *POST* http://localhost:8000/api/orders/  
    *Content-Type*: application/json  
      
    {  
     "user": 4,  
     "products": [  
     {  
     "product": 2,  
     "quantity": 3  
     },  
     {  
     "product": 5,  
     "quantity": 1  
     }  
     ]  
    }
  + We can create a order with creating products and quantity in through table when creating the order itself
* GET /api/orders/<pk>/ — Get single order ✅
* PUT /api/orders/<pk>/ — Update order ✅
* DELETE /api/orders/<pk>/ — Delete order ✅

All tests passed successfully!

**🔹 g) Git Clean Up and Commit**

* Updated .gitignore to ignore .pyc, \_\_pycache\_\_, .venv/, etc.
* Removed cached unnecessary files using:

bash

CopyEdit

1. git rm --cached -r \*.pyc
2. git rm --cached -r \_\_pycache\_\_/
3. git rm --cached -r .venv/

* Staged everything properly:

bash

CopyEdit

git add .

* Committed with a meaningful message:

bash

CopyEdit

git commit -m "Added Order service and cleaned git repo"

* Pushed to GitHub:

bash

CopyEdit

git push origin main

**🎯 Conclusion**

Now we have a fully working **Order Management service** with clean code, proper APIs, and ready for production deployment (Elastic Beanstalk or other platforms).

**✅ Deliverables Summary:**

| **File** | **Purpose** |
| --- | --- |
| models.py | Order & OrderProduct models |
| serializers.py | Serialize Order and related models |
| views.py | Class-based API Views |
| urls.py | API Endpoints |
| populate.py | Dummy Data Population Script |
| .gitignore | Clean Git management |

**2. Detailed Models**

**a. Product Model**

**Represents individual items available for purchase.**

**Fields**

* **name: Name of the product (e.g., Laptop, Phone).**
* **price: Decimal field for the current selling price.**
* **stock\_quantity: Tracks the number of units available in inventory.**

**Key Features**

1. **Stock Tracking: Prevents overselling by validating available stock.**
2. **Flexibility: Allows for dynamic updates to product attributes, like price and description.**

**Example**

**laptop = Product.objects.create(name="Laptop", price=50000, stock\_quantity=10)**

**b. Order Model**

**Represents a customer's order and tracks the lifecycle, including financial calculations.**

**Fields**

* **user: Foreign key linking the order to a customer.**
* **subtotal: Total cost of all items before adjustments.**
* **discount: Absolute discount applied to the order.**
* **tax\_amount: Tax applied to the order.**
* **shipping\_cost: Shipping charges for the order.**
* **total: Final payable amount after adjustments.**
* **status: Tracks the order's progress (e.g., Draft, Confirmed, Delivered).**

**Methods**

1. **clean(): Validates business rules before saving the order, such as ensuring it contains products and checking discount limits.**
2. **calculate\_totals(): Calculates subtotal and total fields based on associated line items.**

**example**

* **order = Order.objects.create(user=user)**
* **order.discount = Decimal('5000') # Apply a discount**
* **order.tax\_amount = Decimal('2345') # Apply tax**
* **order.shipping\_cost = Decimal('500') # Add shipping cost**
* **order.calculate\_totals()**
* **print(order.total) # Final payable amount**

**c. OrderProduct Model**

Intermediary model linking Order and Product. Represents individual line items in the order.

**Fields**

* order: Foreign key linking the line item to a parent order.
* product: Foreign key linking the line item to a specific product.
* quantity: Number of units ordered.
* price\_at\_purchase: Freezes the product price at the time of order creation.

**Methods**

1. clean(): Ensures sufficient stock is available for the requested quantity.
2. save(): Captures the price at purchase and updates the parent order’s totals.

**Example**

* + order\_product = OrderProduct(order=order, product=laptop, quantity=2)
  + order\_product.save()
  + print(order\_product.price\_at\_purchase) # Output: 50000 (price frozen at time of purchase)

**3. Scenarios and Examples**

**Scenario 1: Empty Order Creation**

**Example**

* + **order = Order.objects.create(user=user)**
  + **print(order.subtotal) # Output: 0 (no products added yet)**

**Behavior**

* **The order is saved with default values (subtotal = 0, total = 0) until products are added.**

**Scenario 2: Adding Products to an Order**

**Example**

* **order\_product = OrderProduct(order=order, product=laptop, quantity=2)**
* **order\_product.save()**
* **print(order.subtotal) # Output: 100000 (2 laptops × ₹50000)**

**Behavior**

* **The subtotal is updated based on the line totals from OrderProduct.**

**Scenario 3: Handling Validation Failures**

**Example**

* **order\_product = OrderProduct(order=order, product=laptop, quantity=15) # Exceeds stock**
* **order\_product.save() # Raises ValidationError: "Only 10 available in stock"**

**Behavior**

* **Validation prevents saving a line item that exceeds available stock.**

**Scenario 4: Price Freeze**

**Example**

* **order\_product = OrderProduct(order=order, product=laptop, quantity=1)**
* **order\_product.save()**
* **# Update the product price**
* **laptop.price = 55000**
* **laptop.save()**
* **print(order\_product.price\_at\_purchase) # Output: 50000 (price frozen at time of purchase)**

**Behavior**

* **The price\_at\_purchase remains fixed, even if the product price changes later.**

**4. Test Cases**

**Validation Test: Stock**

**Test Code**

* **order\_product = OrderProduct(order=order, product=laptop, quantity=15)**
* **try:**
  + **order\_product.save()**
* **except ValidationError as e:**
  + **print(e) # Expected: "Only 10 available in stock"**

**Workflow Test: Status Transition**

**Test Code**

* **order.status = 'delivered' # Transition to terminal status**
* **order.save()**
* **order.status = 'processing' # Invalid transition from terminal**
* **try:**
  + **order.save()**
* **except ValidationError as e:**
  + **print(e) # Expected: "Cannot change status from Delivered"**

**Financial Calculation Test**

**Test Code**

* **order.discount = Decimal('5000')**
* **order.tax\_amount = Decimal('2345')**
* **order.shipping\_cost = Decimal('500')**
* **order.calculate\_totals()**
* **print(order.total) # Expected: Subtotal - Discount + Tax + Shipping**

**5. Advanced Concepts**

**Performance Optimization**

* **Use select\_related to reduce the number of queries when fetching related data:**

**order\_products = OrderProduct.objects.select\_related('product').filter(order=order)**

**Handling Empty Orders**

* **Automatically delete empty orders after a grace period:**
  + **if not order.order\_products.exists():**
    - **order.delete()**

**Detailed Step-by-Step Flow:**

1. **Order Creation**

**order = Order.objects.create(user=user)**

* + **Generates empty order with default status=DRAFT**
  + **First save creates PK and timestamps**

1. **Adding Products**

**OrderProduct.objects.create(order=order, product=product1, quantity=2)**

* + **Each product addition triggers:  
    a. OrderProduct.clean() validates stock availability  
    b. price\_at\_purchase is frozen  
    c. Line item saved**

1. **Order Recalculation**

**# Automatic via OrderProduct.post\_save signal:**

**order.calculate\_totals()**

* + **Sums all line\_total() from OrderProducts**
  + **Applies discount, tax, shipping**
  + **Ensures non-negative total**

1. **Order Validation**

**order.full\_clean() # Explicit or implicit via save()**

* + **Validates:**
    - **At least one product exists**
    - **Discount ≤ subtotal**
    - **Valid status transition**

1. **Status Handling**

**if status in TERMINAL\_STATUSES:**

**order.completed\_at = timezone.now()**

* + **Updates completed\_at when reaching terminal states**
  + **Prevents changes from terminal states**

1. **Final Save**

**order.save() # Within transaction.atomic()**

* + **All changes committed atomically**
  + **Updates all calculated fields**

**This flow ensures:**

* **Data consistency through atomic transactions**
* **Price integrity via frozen prices**
* **Valid state transitions**
* **Accurate financial calculations**
* **Proper inventory checks**

**Would you like me to provide any specific part of this flow in more detail?**

**Order Management System - Documentation**

**1. Overview**

**The Order Management System is a Django REST-based API that handles order creation, retrieval, updating, and deletion. It integrates with a product system to manage orders with products, calculates totals (subtotal, tax, shipping, discounts), and enforces business rules.**

**Key Features**

**✅ Order CRUD Operations – Create, Read, Update, Delete orders  
✅ Product Management – Associate products with quantities in orders  
✅ Financial Calculations – Automatic subtotal, tax, shipping, and discount calculations  
✅ Status Workflow – Track order status (pending, confirmed, delivered, cancelled, returned)  
✅ Validation & Error Handling – Ensures business rules (e.g., no negative totals, valid status transitions)  
✅ Optimized Queries – Uses select\_related and prefetch\_related for performance**

**2. System Architecture**

**Layered Architecture**

* **Models (models.py)**
  + **Order: Core order details (user, status, financials)**
  + **OrderProduct: Junction table for products in an order (with frozen prices)**
  + **AuditData: Abstract model for created\_at and updated\_at timestamps**
* **Services (services.py)**
  + **Business logic layer (calculations, status updates, inventory checks)**
* **Views (views.py)**
  + **API endpoints (RESTful structure)**
  + **Uses APIView for explicit HTTP method handling**
* **Serializers (serializers.py)**
  + **Data validation & transformation**
  + **Handles nested product data**
* **URLs (urls.py)**
  + **Clean routing to views**

**3. API Endpoints**

| **Endpoint** | **Method** | **Description** | **Requires** |
| --- | --- | --- | --- |
| **/orders/** | **GET** | **List all orders** | **-** |
| **/orders/** | **POST** | **Create a new order** | **user\_id, products (list)** |
| **/orders/{pk}/** | **GET** | **Retrieve a single order** | **order\_id** |
| **/orders/{pk}/** | **PATCH** | **Partially update an order (status/products)** | **order\_id, optional updates** |
| **/orders/{pk}/** | **DELETE** | **Delete an order** | **order\_id** |

**4. Business Rules**

**✔ No Negative Totals – subtotal, discount, total cannot be negative.  
✔ Valid Status Transitions – E.g., cannot move from cancelled back to pending.  
✔ Minimum Order Quantity – Each product must have quantity >= 1.  
✔ Discount Limit – discount cannot exceed subtotal.**

**6. Future Improvements**

**🔹 Pagination – For large order lists (/orders/?page=2).  
🔹 Search/Filtering – By status, date, user.  
🔹 Authentication – JWT/OAuth2 for secure access.  
🔹 Webhooks – Notify other services on order updates.**

* **This system provides a scalable, maintainable, and well-documented API for managing orders. It follows REST principles, ensures data consistency, and is ready for extension**

**eCommerce Backend Project - Full Explanation (Django + DRF)**

**Project Overview**

**This project is an eCommerce backend built using Django and Django REST Framework. It manages Users, Products, Categories, Orders, and tracks Order Items (products within an order).**

**The architecture focuses on:**

* **Clean database modeling (with proper relationships)**
* **RESTful APIs for frontend consumption**
* **Flexible handling of taxes, discounts, and order totals**
* **Maintainable and scalable design**

**Models Explained**

**1. AuditData (Abstract Model)**

* **Common fields for tracking:**
  + **created\_at: When the record was created.**
  + **updated\_at: When the record was last updated.**
* **Abstract: It is not a database table itself, but other models inherit it.**

**2. User**

* **Fields: name**
* **Relation:**
  + **One User can place multiple Orders.**
* **String Representation: Returns the user's name for easier admin usage.**

**3. Category**

* **Fields: name**
* **Relation:**
  + **One Category can have multiple Products.**
* **String Representation: Category name.**

**4. Products**

* **Fields:**
  + **name, price, description, is\_available**
  + **category: Foreign Key to Category**
* **Relation:**
  + **A product can belong to one category.**
  + **A product can be part of many orders.**
* **Meta:**
  + **verbose\_name\_plural set to "Products" for admin panel.**
* **String Representation: "Product name - ₹Price"**

**5. Order**

* **Fields:**
  + **user: ForeignKey to User**
  + **products: ManyToMany with Products (through OrderProduct)**
  + **order\_status: Pending / Success / Failed**
* **Methods:**
  + **total\_bill(discount, tax): Calculates final payable amount.**
* **String Representation: Order summary with User name and Product list.**

**6. OrderProduct (Through Table)**

* **Tracks quantity of each product in an order.**
* **Fields:**
  + **order: FK to Order**
  + **product: FK to Products**
  + **quantity**
* **String Representation: "Order #id - Product x Quantity"**

**Serializers Explained**

**1. ProductSerializer**

* **Only exposes id, name, price for efficient transmission.**

**2. OrderProductSerializer**

* **Nested structure showing product details + quantity inside an order.**

**3. OrderSerializer**

* **Dual purpose:**
  + **Input: products field (list of dicts)**
  + **Output: order\_products nested serializer**
  + **Dynamic Calculation: total\_payable\_amount**
* **Custom create() method:**
  + **Accepts product list.**
  + **Creates an Order.**
  + **Links each product to the order using OrderProduct.**
* **get\_total\_payable\_amount():**
  + **Accepts optional tax and discount through query parameters.**

**Views Explained**

**1. greet(request)**

* **Quick test function.**
* **Updates the name of the first Product.**

**2. create\_r\_get\_products(request)**

* **Handles both GET (list products) and POST (create a product).**

**3. get\_product(request, id)**

* **Fetches details of a single product by ID.**

**4. filter\_products(request)**

* **Supports filtering products based on:**
  + **Name (partial match)**
  + **Description**
  + **Min Price / Max Price**

**5. update\_product(request, id)**

* **PATCH endpoint to update specific fields (e.g., price).**

**6. delete\_product(request, id)**

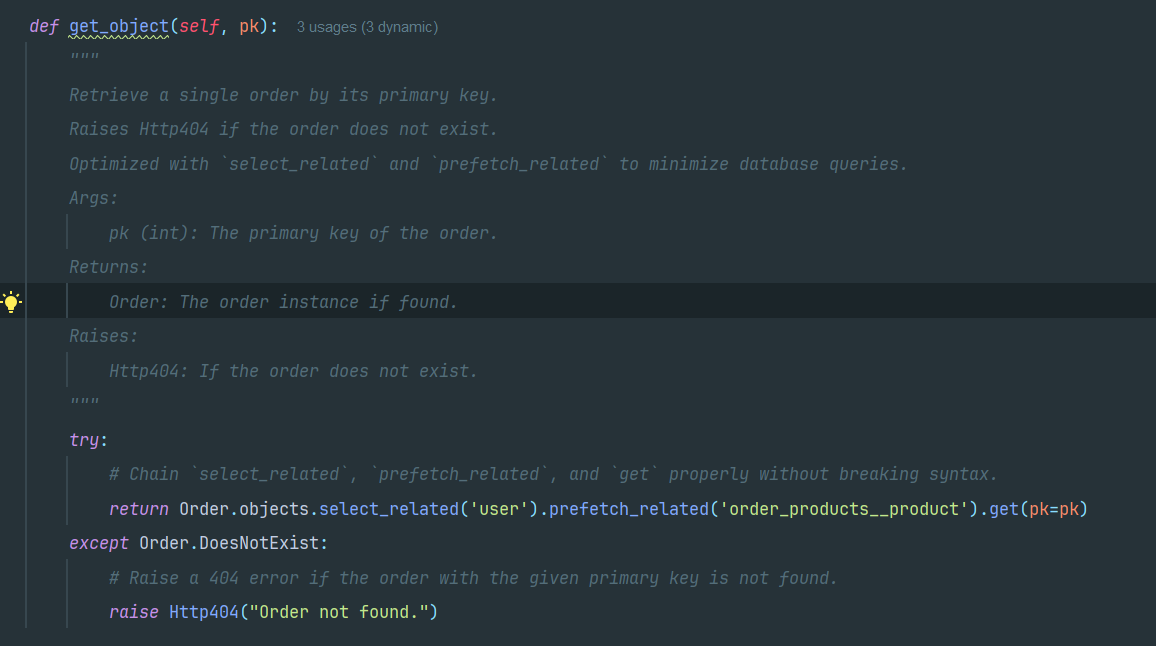
* **Deletes a product by ID.**

**7. OrderListCreateView (CBV)**

* **GET: List all orders**
* **POST: Create new order**

**8. OrderDetailView (CBV)**

* **GET: Retrieve one order**
* **PATCH: Update order status**
* **DELETE: Delete order**

****

**Overview of the Code**

This method retrieves an Order by its primary key (pk). It:

1. **Optimizes Query Performance** using:
   1. select\_related('user'): Joins the user table in the same query to fetch related user details.
   2. prefetch\_related('order\_products\_\_product'): Preloads order\_products and the associated product objects in an efficient query
2. **Handles Missing Orders Gracefully**:
   1. If no Order matches the given pk, it raises an Http404 error with a meaningful message ("Order not found").

**Order.objects.select\_related('user').prefetch\_related('order\_products\_\_product').get(pk=pk)**

**select\_related('user'):**

* **Joins the User table to fetch the Order and its related user in a single query.**
* **Example query:**

**SELECT \* FROM order**

**INNER JOIN user ON order.user\_id = user.id**

**WHERE order.id = 1;**

**prefetch\_related('order\_products\_\_product'):**

* **Fetches all OrderProduct records related to the order and their associated Product objects in one additional query.**
* **Example queries:**
  1. **Fetch order products:**

**SELECT \* FROM order\_product WHERE order\_id = 1;**

* 1. **Fetch associated products:**

**SELECT \* FROM product WHERE id IN (1, 2);**

* 1. **Step 3: Return the Result**
     + If the order with pk=1 exists, this method returns the following **optimized result**:

**Order(**

**pk=1,**

**user=User(pk=1, username="Lakshmi"),**

**created\_at="2023-01-01",**

**order\_products=[**

**OrderProduct(order=..., product=Product(pk=1, name="Laptop"), quantity=2),**

**OrderProduct(order=..., product=Product(pk=2, name="Smartphone"), quantity=1)**

**]**

**)**

**Important Concepts Used**

| **Concept** | **Explanation** |
| --- | --- |
| **@api\_view** | **Function-based views for basic CRUD operations** |
| **APIView** | **Class-based views for cleaner structure** |
| **csrf\_exempt** | **Skip CSRF checks during simple API testing** |
| **Q Objects** | **Flexible filtering queries** |
| **partial=True** | **Allow partial updates during PATCH** |
| **Response** | **DRF's formatted JSON responses** |
| **status** | **HTTP response codes (200, 201, 404, etc.)** |

**Project Dataflow Diagram**

**Products APIs**

**|**

**|-- GET /products/ --> List all products**

**|-- POST /products/ --> Create product**

**|-- GET /products/<id>/ --> Get one product**

**|-- PATCH /products/<id>/ --> Update product**

**|-- DELETE /products/<id>/ --> Delete product**

**|-- GET /products/filter/ --> Filter products**

**Orders APIs**

**|**

**|-- GET /orders/ --> List all orders**

**|-- POST /orders/ --> Create a new order**

**|-- GET /orders/<id>/ --> Get one order**

**|-- PATCH /orders/<id>/ --> Update order status**

**|-- DELETE /orders/<id>/ --> Delete order**

**Final Notes**

* **Modular, clean, extendable.**
* **Supports flexible billing with taxes and discounts.**
* **Proper error handling.**
* **Ready for production extension (like authentication, payments, etc.).**

**Saved for you to build bigger projects easily 🚀📁**

<https://chatgpt.com/canvas/shared/680e786f857481919eeca6cdfc24d3be>

┌───────────────────────┐

│ │

│ Order Creation │

│ (Order.objects. │

│ create()) │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Initial Save │

│ - Generates PK │

│ - Sets created\_at │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Add OrderProducts │

│ (via bulk\_create or │

│ individual saves) │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ OrderProduct.save() │

│ - Validates stock │

│ - Freezes price │

│ - Saves line item │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Trigger Order │

│ Recalculation │

│ - calculate\_totals() │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Order.save() │

│ - clean() validation │

│ - Status transitions │

│ - Terminal state │

│ handling │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Final Calculations │

│ - subtotal │

│ - discount │

│ - tax │

│ - shipping │

│ - total │

│ │

└──────────┬────────────┘

│

▼

┌───────────────────────┐

│ │

│ Database Commit │

│ (transaction atomic) │

│ │

└───────────────────────┘