Backend Development Roadmap - Next Steps (3-5 Days)

Project: The Petal Pouches E-commerce Platform

Phase: Backend Core Development (Post-Cloudinary Integration)

Timeline: Days 1-5

Date: October 19, 2025

📊 Current Status Overview

✅ Completed:

- Cloudinary image upload integration
- Product creation with image upload to Cloudinary
- Basic product listing (GET all products)
- Supabase database connection
- Multer file upload middleware
- Basic server setup with Express
- Environment configuration

⏳ In Progress:

- Product update functionality (scaffolded, needs testing)
- Product delete functionality (scaffolded, needs testing)

❌ Not Started:

- Category management system
- Product variants system
- Individual product retrieval
- Advanced product filtering
- Image gallery management
- Inventory management

🎯 Development Strategy

Why This Order?

1. **Categories First** - Products reference categories; need valid category_id values
2. **Complete Products** - Finish CRUD before adding complexity (variants, bundles)
3. **Variants System** - Enable product options (colors, sizes, materials)
4. **Quality Assurance** - Test thoroughly before moving to customer-facing features

What Comes After (Days 6+):

- Shopping cart system
- Order management
- Payment integration (Razorpay/Stripe)
- Shipping integration (Shiprocket)
- User authentication
- Admin authentication & authorization

📅 Day-by-Day Breakdown

DAY 1: Category Management System

Objective:

Build complete category CRUD operations to enable proper product categorization.

Why Categories First?

- Products table has `category_id` foreign key to Categories table
- Admin needs to create categories before assigning products to them
- Frontend will need categories for filtering and navigation

Tasks for Day 1:

Task 1.1: Create Category Controller

File to Create: `backend/src/controllers/categoryController.js`

Functions to Implement:

1. `createCategory`
   - Purpose: Admin creates new product category
   - Method: POST
   - Inputs: name, description
   - Validations:
     - Name is required and unique
     - Description is optional
   - Returns: Created category object with UUID
2. `getAllCategories`
   - Purpose: Fetch all categories (public route)
   - Method: GET
   - Query params: None initially (later add pagination)
   - Returns: Array of all categories
3. `getCategoryById`
   - Purpose: Get single category details
   - Method: GET
   - Params: category UUID
   - Returns: Category object with id, name, description, created_at
4. `updateCategory`
   - Purpose: Admin updates category name/description
   - Method: PUT
   - Params: category UUID
   - Inputs: name (optional), description (optional)
   - Validations: At least one field must be updated
   - Returns: Updated category object
5. `deleteCategory`
   - Purpose: Admin deletes category

- Method: DELETE
- Params: category UUID
- Important: Check if products exist in this category
- Options:
    - Prevent deletion if products exist (recommended)
    - Or set product category_id to NULL (optional)
- Returns: Success message

Error Handling Required:

- Category not found (404)
- Duplicate category name (409)
- Database errors (500)
- Validation errors (400)

Task 1.2: Create Category Routes

File to Create: `backend/src/routes/categories.js`

Routes to Define:

Public Routes:

```
GET    /api/categories          - Get all categories
GET    /api/categories/:id      - Get single category
```

Admin Routes:

```
POST   /api/admin/categories     - Create category
PUT    /api/admin/categories/:id - Update category
DELETE /api/admin/categories/:id - Delete category
```

Middleware:

- Admin routes: Will need authentication middleware (implement later)
- For now: Create routes without auth, add auth in Phase 2

Task 1.3: Update Main Server File

File to Modify: `backend/src/index.js`

Changes:

- Import category routes
- Register routes: `app.use('/api/categories', categoryRoutes)`
- Ensure admin routes are also registered properly

Task 1.4: Testing

Tools: Postman, Thunder Client, or Insomnia

Test Cases:

1. **Create Category**
   - Test with valid data (name + description)
   - Test with only name (description optional)
   - Test with duplicate name (should fail)
   - Test with empty name (should fail)
2. **Get All Categories**
   - Verify returns empty array if no categories
   - Verify returns all created categories
   - Check correct format (id, name, description, created_at)
3. **Get Single Category**
   - Test with valid category UUID
   - Test with invalid UUID (should return 404)
4. **Update Category**
   - Update name only
   - Update description only
   - Update both fields
   - Test with invalid UUID (should return 404)

5. **Delete Category**
    - Delete category with no products (should succeed)
    - Delete category with products (decide behavior)
    - Test with invalid UUID (should return 404)

Task 1.5: Seed Sample Categories

Purpose: Create initial categories for testing product creation

Recommended Categories for The Petal Pouches:

1. Necklaces
2. Rings
3. Bracelets & Bangles
4. Earrings
5. Anklets
6. Soft Toys & Plushies
7. Hair Accessories
8. Phone Cases
9. Bags & Pouches
10. Stationery & Gifts

Method:

- Use Postman/Thunder Client to POST each category
- Or create a seed script: `backend/scripts/seedCategories.js`
- Save the returned UUIDs for product creation

Day 1 Deliverables:

- ✅ `categoryController.js` created and working
- ✅ `categories.js` routes created
- ✅ Server updated with category routes

- ✅ All 5 CRUD operations tested
- ✅ 10 sample categories created in database
- ✅ Category UUIDs documented for product assignment

DAY 2: Complete Product Management

Objective:

Finish the product CRUD system and add advanced features.

Tasks for Day 2:

Task 2.1: Add Get Product by ID

File to Modify: `backend/src/controllers/productController.js`

New Function: **`getProductById`**

- Purpose: Fetch single product details for Product Detail Page (PDP)
- Method: GET
- Route: /api/products/:id
- Params: product UUID
- Returns: Complete product object including:
  - All product fields
  - Category details (join with Categories table)
  - Image URL from Cloudinary
  - Stock information
- Error handling: 404 if product not found

Task 2.2: Test Update Product

File Already Exists: `backend/src/controllers/productController.js`

Function: **`updateProduct`** (already scaffolded)

Testing Scenarios:

1. **Update Product Details Only (No Image Change)**
    - Update title
    - Update description
    - Update price
    - Update stock
    - Update category_id (to valid category UUID)
2. **Update Product with New Image**
    - Upload new image to Cloudinary
    - Update img_url in database
    - **Important:** Delete old image from Cloudinary
    - Extract publicId from old URL
    - Call `deleteFromCloudinary(publicId)`
3. **Update Some Fields, Keep Others**
    - Partial updates should work
    - Fields not sent should remain unchanged

Edge Cases to Test:

- Invalid product UUID (404)
- Invalid category_id (foreign key error)
- Image upload fails (should rollback)
- Very large images (should handle file size limit)

Task 2.3: Test Delete Product

File Already Exists: `backend/src/controllers/productController.js`

Function: **`deleteProduct`** (already scaffolded)

Testing Scenarios:

1. **Basic Delete**

- Delete product with valid UUID
- Verify product removed from database
- Verify image deleted from Cloudinary

2. **Extract publicId from Cloudinary URL**
   - URL format: `https://res.cloudinary.com/drmza0a9d/image/upload/v1234567890/products/abc123.jpg`
   - publicId: `products/abc123`
   - Implement helper function: `extractPublicIdFromUrl(url)`
3. **Handle Delete Failures**
   - Product has active orders (decide: prevent delete or soft delete)
   - Product not found (404)
   - Cloudinary delete fails (log error but don't fail request)

Implementation Decision:

- **Soft Delete** (Recommended): Add `is_deleted` boolean to Products table
- **Hard Delete**: Actually remove from database (simpler for MVP)

Task 2.4: Add Advanced Filtering to getAllProducts

File to Modify: `backend/src/controllers/productController.js`

Function: **`getAllProducts`** (enhance existing)

New Query Parameters to Support:

1. **Category Filter**
   - Query param: `?category_id=uuid`
   - Filter products by category
2. **Price Range Filter**
   - Query params: `?min_price=500&max_price=2000`
   - Filter products within price range
3. **Search by Title**
   - Query param: `?search=necklace`

- Case-insensitive search in product title

4. **Sort Options**
   - Query param: `?sort=price_asc` or `?sort=price_desc`
   - Sort by: price (ascending/descending), created_at (newest first)
5. **Pagination**
   - Query params: `?page=1&limit=20`
   - Default: page=1, limit=20
   - Returns: products array + metadata (totalCount, totalPages, currentPage)
6. **Stock Filter**
   - Query param: `?in_stock=true`
   - Filter only products with stock > 0

Example Combined Query:

```
GET /api/products?
category_id=uuid&min_price=500&max_price=2000&search=pink&sort=price_asc&page=1&limit=20
```

Implementation Notes:

- Use Supabase query builder `.filter()`, `.gte()`, `.lte()`, `.ilike()`
- Build query conditionally based on provided params
- Add proper error handling for invalid params

Task 2.5: Add Product Routes

File to Modify: `backend/src/routes/products.js`

New Route to Add:

```
GET /api/products/:id  - Get single product
```

Verify Existing Routes:

```
GET    /api/products          - Get all products (with filters)
POST   /api/admin/products    - Create product (already working)
PUT    /api/admin/products/:id    - Update product
DELETE /api/admin/products/:id    - Delete product
```

Task 2.6: Create Helper Utilities

File to Create: `backend/src/utils/cloudinaryHelpers.js`

Functions:

1. **`extractPublicIdFromUrl(cloudinaryUrl)`**
    - Extracts publicId from full Cloudinary URL
    - Example: `https://res.cloudinary.com/.../products/abc.jpg` → `products/abc`
2. **`validateImageUrl(url)`**
    - Check if URL is from Cloudinary domain
    - Validate URL format
3. **`getOptimizedUrl(publicId, transformations)`**
    - Generate optimized Cloudinary URLs
    - For thumbnails, different sizes, formats

Day 2 Deliverables:

- ✅ `getProductById` implemented and tested
- ✅ `updateProduct` fully tested (with/without image)
- ✅ `deleteProduct` fully tested (with Cloudinary cleanup)
- ✅ Advanced filtering implemented in `getAllProducts`
- ✅ Helper utilities created for Cloudinary operations
- ✅ All product routes working and documented

DAY 3: Product Variants System

Objective:

Enable products to have variants (different colors, sizes, materials) as per database schema.

Background:

According to your schema:

- `products` table has `has_variants` boolean field
- `product_variants` table stores variant details (SKU, attributes, price, stock, image)
- Variants have JSONB `attributes` field: `{"color": "Pink", "size": "Small"}`

Tasks for Day 3:

Task 3.1: Understand Variant Architecture

Use Cases:

1. **Product WITHOUT Variants** (Simple Product)
   - Example: "Custom Name Necklace" - one size, no color options
   - `has_variants = false`
   - Stock and price stored in `products` table
2. **Product WITH Variants** (Variable Product)
   - Example: "Heart Ring" - available in Gold/Silver, sizes 6/7/8
   - `has_variants = true`
   - Each variant has its own SKU, price, stock, image
   - Variants stored in `product_variants` table

Variant Attributes Examples:

- Jewelry: `{"metal": "Gold", "size": "7"}`
- Soft Toys: `{"color": "Pink", "size": "Medium"}`
- Accessories: `{"color": "Blue", "pattern": "Floral"}`

Task 3.2: Create Variant Controller

File to Create: `backend/src/controllers/variantController.js`

Functions to Implement:

1. **`createVariant`**
   - Purpose: Admin adds variant to existing product
   - Method: POST
   - Route: /api/admin/products/:productId/variants
   - Required Fields:
     - `product_id` (from URL param)
     - `sku` (unique variant SKU)
     - `attributes` (JSONB: color, size, etc.)
     - `price` (variant-specific price, optional)
     - `stock` (variant stock quantity)
   - Optional Fields:
     - `weight` (for shipping calculations)
     - `img_url` (variant-specific image)
     - `is_default` (default variant for product)
   - Validations:
     - Product must exist
     - Product must have `has_variants = true`
     - SKU must be unique
     - Attributes must be valid JSONB
   - Returns: Created variant object
2. **`getVariantsByProductId`**
   - Purpose: Fetch all variants for a specific product
   - Method: GET
   - Route: /api/products/:productId/variants
   - Returns: Array of variants with all fields
   - Use case: Product Detail Page shows all available options
3. **`getVariantById`**

- Purpose: Get single variant details
- Method: GET
- Route: /api/variants/:variantId
- Returns: Single variant object
- Use case: Cart/Order needs specific variant info

4. `updateVariant`
   - Purpose: Admin updates variant details
   - Method: PUT
   - Route: /api/admin/variants/:variantId
   - Updatable Fields:
     - SKU
     - Attributes (JSONB)
     - Price
     - Stock
     - Weight
     - Image URL
     - is_default status
   - Returns: Updated variant object
5. `deleteVariant`
   - Purpose: Admin removes variant
   - Method: DELETE
   - Route: /api/admin/variants/:variantId
   - Important Checks:
     - Check if variant is in active orders
     - Prevent deletion if last variant (product must have at least 1 variant)
   - Optional: Delete variant image from Cloudinary
   - Returns: Success message
6. `updateVariantStock`

- Purpose: Update stock quantity (separate from full update)
- Method: PATCH
- Route: /api/admin/variants/:variantId/stock
- Body: `{ stock: 50 }`
- Use case: Quick stock adjustments
- Returns: Updated variant with new stock count

Task 3.3: Create Variant Routes

File to Create: `backend/src/routes/variants.js`

Public Routes:

```
GET /api/products/:productId/variants  - Get all variants for product
GET /api/variants/:variantId           - Get single variant details
```

Admin Routes:

```
POST   /api/admin/products/:productId/variants  - Create variant
PUT    /api/admin/variants/:variantId           - Update variant
PATCH  /api/admin/variants/:variantId/stock     - Update stock only
DELETE /api/admin/variants/:variantId           - Delete variant
```

Task 3.4: Update Product Controller for Variants

File to Modify: `backend/src/controllers/productController.js`

Changes Needed:

1. **Modify `createProduct`**
   - Add `has_variants` field to insert
   - Default: `has_variants = false`
2. **Modify `getProductById`**
   - If product has variants (`has_variants = true`), fetch all variants
   - Return product object with nested `variants` array

- Example response:

```json
{
 "id": "uuid",
 "title": "Heart Ring",
 "has_variants": true,
 "variants": [
   {
    "id": "variant-uuid-1",
    "sku": "RING-GOLD-7",
    "attributes": {"metal": "Gold", "size": "7"},
    "price": 1299,
    "stock": 10
   },
   {
    "id": "variant-uuid-2",
    "sku": "RING-SILVER-7",
    "attributes": {"metal": "Silver", "size": "7"},
    "price": 999,
    "stock": 15
   }
 ]
}
```

3. **Modify `getAllProducts`**
   - Option 1: Include variant count in product list
   - Option 2: Don't fetch variants in list view (only in detail view)
   - Recommended: Option 2 for performance

Task 3.5: Create Variant Image Upload Endpoint

Purpose: Upload variant-specific images (e.g., ring in gold vs silver)

File to Modify: `backend/src/routes/admin.js`

New Route:

```
POST /api/admin/variants/:variantId/image
```

Implementation:

- Use same Multer middleware as product images
- Upload to Cloudinary in `products/variants/` folder
- Update variant's `img_url` field
- Delete old variant image if exists

Task 3.6: Testing Variant System

Test Scenarios:

1. **Create Product WITH Variants**
   - Create base product with `has_variants = true`
   - Create 3-5 variants with different attributes
   - Verify each variant has unique SKU
   - Verify JSONB attributes stored correctly
2. **Create Product WITHOUT Variants**
   - Create base product with `has_variants = false`
   - Attempt to create variant (should fail or warn)
   - Stock/price stored in product table
3. **Get Product with Variants**
   - Fetch product by ID
   - Verify variants array included
   - Check all variant details present
4. **Update Variant**
   - Update price
   - Update stock
   - Update attributes (change color/size)
   - Upload variant-specific image
5. **Delete Variant**
   - Delete variant with no orders

    - Attempt to delete last variant (should fail)
    - Attempt to delete variant in active order (should prevent)

6. **Stock Management**
    - Update variant stock via PATCH endpoint
    - Verify stock decrements after order (implement later)

Day 3 Deliverables:

- ✅ `variantController.js` created with all CRUD functions
- ✅ `variants.js` routes created and registered
- ✅ Product controller updated to handle variants
- ✅ Variant image upload endpoint created
- ✅ Comprehensive variant testing completed
- ✅ Sample products with variants created for testing

DAY 4: Testing, Error Handling & Validation

Objective:

Ensure all backend endpoints are robust, secure, and production-ready.

Tasks for Day 4:

Task 4.1: Comprehensive API Testing

Tool Setup:

- Use Postman or Thunder Client
- Create organized collection with folders:
    - Categories
    - Products
    - Variants
    - Admin Operations

Test Each Endpoint:

1. **Happy Path Tests**
   - All endpoints work with valid data
   - Correct status codes (200, 201, 204)
   - Correct response format
2. **Error Path Tests**
   - Invalid UUIDs (404)
   - Missing required fields (400)
   - Duplicate entries (409)
   - Foreign key violations (400)
   - Invalid data types (400)
3. **Edge Cases**
   - Very long text inputs
   - Special characters in names
   - Zero/negative prices
   - Negative stock quantities
   - Empty arrays/objects
   - Null values where not allowed
4. **Performance Tests**
   - Large product lists (pagination)
   - Multiple filters applied
   - Response times under 500ms

Task 4.2: Add Input Validation

File to Create: `backend/src/middlewares/validation.js`

Validation Rules to Implement:

1. **Product Validation**
   - Title: required, string, 3-200 characters

- Description: optional, string, max 2000 characters
- Price: required, positive integer
- Stock: required, non-negative integer
- SKU: required, string, unique, 3-50 characters
- Category_id: optional, valid UUID format

2. **Category Validation**
    - Name: required, string, 3-100 characters, unique
    - Description: optional, string, max 500 characters
3. **Variant Validation**
    - SKU: required, string, unique, 3-50 characters
    - Attributes: required, valid JSONB object
    - Price: optional, positive integer
    - Stock: required, non-negative integer
    - Weight: optional, positive number

Validation Library Options:

- Option 1: `express-validator` (popular, express-specific)
- Option 2: `joi` (schema validation)
- Option 3: `zod` (TypeScript-friendly)

Implementation:

- Create validation middleware for each entity
- Apply to routes before controller functions
- Return 400 with clear error messages

Task 4.3: Enhance Error Handling

File to Create: `backend/src/middlewares/errorHandler.js`

Centralized Error Handler:

1. **Error Types to Handle**

- Database errors (Supabase/PostgreSQL)
- Validation errors
- Authentication errors (future)
- Cloudinary upload errors
- Not found errors (404)
- Conflict errors (409)

2. **Error Response Format**

```json
{
  "success": false,
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Product title is required",
    "field": "title"
  }
}
```

3. **Error Logging**
    - Log errors to console (development)
    - Log errors to file (production)
    - Future: Send to error tracking service (Sentry)

Update All Controllers:

- Use centralized error handler
- Throw specific error types
- Include helpful error messages

Task 4.4: Add Request Logging

File to Create: `backend/src/middlewares/logger.js`

Purpose: Log all incoming requests for debugging

Information to Log:

- Request method (GET, POST, etc.)
- Request URL
- Request body (sanitize sensitive data)
- Response status code
- Response time
- IP address
- User agent

Implementation:

- Use `morgan` middleware for HTTP request logging
- Custom middleware for additional logging
- Different log levels: development vs production

Task 4.5: Database Query Optimization

Review and Optimize:

1. **Add Database Indexes**
   - Index on `products.category_id` (foreign key)
   - Index on `products.sku` (unique, frequently queried)
   - Index on `product_variants.product_id` (foreign key)
   - Index on `product_variants.sku` (unique)
   - Index on `categories.name` (unique, frequently queried)
2. **Optimize Queries**
   - Use `.select()` to fetch only needed columns
   - Avoid N+1 queries (fetch related data in single query)
   - Use `.limit()` and `.offset()` for pagination
   - Use `.count()` for total count queries
3. **Add Query Helpers**
   - File: `backend/src/utils/queryHelpers.js`

- Function: `buildProductFilters(queryParams)`
- Function: `paginateResults(query, page, limit)`

Task 4.6: API Documentation

File to Create: `backend/docs/API.md`

Document Each Endpoint:

Format for each endpoint:

```
markdown
```

```
### POST /api/admin/products
**Description:** Create new product with image upload

**Authentication:** Admin only (to be implemented)

**Request:**
- Method: POST
- Content-Type: multipart/form-data
- Body:
  - image (file, required)
  - title (string, required)
  - description (string, optional)
  - price (integer, required)
  - stock (integer, required)
  - sku (string, required)
  - category_id (uuid, optional)

**Response (201):**
{
  "success": true,
  "message": "Product created successfully",
  "data": {
    "id": "uuid",
    "title": "Pink Heart Necklace",
    ...
  }
}

**Errors:**
- 400: Missing required fields
- 409: SKU already exists
- 500: Server error
```

Organize by Resource:

- Categories
- Products
- Variants
- Future: Cart, Orders, Payments

Day 4 Deliverables:

- ✅ Complete API testing with Postman collection
- ✅ Input validation middleware implemented
- ✅ Centralized error handling implemented
- ✅ Request logging added
- ✅ Database queries optimized with indexes
- ✅ API documentation created
- ✅ All endpoints tested for edge cases

DAY 5: Polish, Documentation & Preparation

Objective:

Finalize backend core, create comprehensive documentation, and prepare for frontend integration.

Tasks for Day 5:

Task 5.1: Code Review & Refactoring

Review All Controllers:

- Check for code duplication
- Extract common logic into helper functions
- Ensure consistent naming conventions
- Add JSDoc comments to functions

File Structure Audit:

```
backend/src/
├── config/
│   ├── supabaseClient.js     ✅ Review
│   ├── cloudinary.js         ✅ Review
│   └── multer.js             ✅ Review
├── controllers/
│   ├── productController.js  ✅ Review & refactor
│   ├── categoryController.js ✅ Review & refactor
│   └── variantController.js  ✅ Review & refactor
├── routes/
│   ├── admin.js              ✅ Review
│   ├── products.js           ✅ Review
│   ├── categories.js         ✅ Review
│   └── variants.js           ✅ Review
├── services/
│   └── cloudinaryService.js  ✅ Review
├── middlewares/
│   ├── errorHandler.js       ✅ Review
│   ├── validation.js         ✅ Review
│   └── logger.js             ✅ Review
├── utils/
│   ├── cloudinaryHelpers.js  ✅ Review
│   └── queryHelpers.js       ✅ Review
└── index.js                  ✅ Review
```

Task 5.2: Environment Variables Documentation

File to Update: `backend/.env.example`

Complete List:

```env

```

```
# Server
NODE_ENV=development
PORT=5000
FRONTEND_URL=http://localhost:5173

# Supabase
SUPABASE_URL=your_supabase_url
SUPABASE_SERVICE_ROLE_KEY=your_service_role_key
SUPABASE_ANON_KEY=your_anon_key

# Cloudinary
CLOUDINARY_CLOUD_NAME=your_cloud_name
CLOUDINARY_API_KEY=your_api_key
CLOUDINARY_API_SECRET=your_api_secret

# Future: Payment (Phase 2)
RAZORPAY_KEY_ID=
RAZORPAY_KEY_SECRET=

# Future: Shipping (Phase 2)
SHIPROCKET_API_KEY=
SHIPROCKET_EMAIL=
SHIPROCKET_PASSWORD=
```

Create Setup Guide:

- File: `backend/docs/SETUP.md`
- How to get each credential
- Step-by-step setup instructions
- Common setup issues and solutions

Task 5.3: Database Seeding Scripts

File to Create: `backend/scripts/seedDatabase.js`

Purpose: Quickly populate database with test data

Seed Data to Create:

1. **Categories (10 items)**

  - Necklaces, Rings, Bracelets, Earrings, etc.

2. **Products (20-30 items)**
   - Mix of simple products (no variants)
   - Mix of products with variants
   - Various price points (₹299 - ₹2999)
   - Different categories
3. **Product Variants (40-50 items)**
   - For products with variants
   - Different colors, sizes, materials
   - Realistic stock quantities

Script Features:

- Check if data already exists (don't duplicate)
- Upload sample images to Cloudinary
- Return summary of seeded data
- Option to clear existing data first

Run Command:

```bash
npm run seed
```

Task 5.4: Create API Testing Collection

Tool: Postman or Thunder Client

Collection Structure:

```
The Petal Pouches API/
├── 📁 Categories/
│   ├── Create Category
│   ├── Get All Categories
│   ├── Get Category by ID
│   ├── Update Category
│   └── Delete Category
├── 📁 Products/
│   ├── Create Product (with image)
│   ├── Get All Products
│   ├── Get All Products (with filters)
│   ├── Get Product by ID
│   ├── Update Product (no image)
│   ├── Update Product (with new image)
│   └── Delete Product
├── 📁 Variants/
│   ├── Create Variant
│   ├── Get Product Variants
│   ├── Get Variant by ID
│   ├── Update Variant
│   ├── Update Variant Stock
│   └── Delete Variant
└── 📁 Tests/
    ├── Error Tests (404, 400, 409)
    └── Edge Cases
```

Export Collection:

- Save as JSON file
- Add to repo: `backend/postman/`
- Include environment variables file

Task 5.5: Performance & Security Audit

Performance Checks:

1. **Response Times**
    - All endpoints under 500ms
    - Image upload under 3 seconds
    - Pagination working efficiently

2. **Database Queries**
    - No N+1 query problems
    - Proper use of indexes
    - Efficient joins
3. **File Uploads**
    - Multer