Project Overview: Greatkart E-commerce

Background of the problem

- Domain: E■commerce: online retail platform for product discovery, cart management, checkout, and order processing.
- Context: Small/medium sellers need an approachable web store with product variations, user accounts, and order handling.
- Pain Points: Fragmented product discovery, manual order tracking, poor UX for variations/attributes, difficulty managing anonymous carts and user carts on login.
- Why this project: Demonstrates a complete Django-based solution (catalog, cart, checkout, admin) that addresses those pain points in a learning/demo-ready codebase.

Scope of the Project

- In■Scope:
- Product catalog with categories, variations, galleries, and reviews.
- Shopping cart (session-based for guests and user-linked for authenticated users).
- User accounts with email verification, profile management, order history.
- Order placement, payment handling stub, and order administration.
- Admin customizations for easier product/order management (thumbnails, inline edits).
- Out■of■Scope:
- Production payment gateway integration (project uses a simplified/payments placeholder).
- Multi-tenant/multi-vendor features.
- Automated deployment pipelines and production caching setups.
- Assumptions: Local SQLite database for dev, static/media served by Django during development, email via dev SMTP or console for verification.

Problem statement

- Core Problem: Build a maintainable web application that enables browsing a product catalog, selecting variations, managing a persistent cart across auth state, and placing orders while providing admins easy product/order management.
- Constraints: Use Django MVT, keep setup reproducible, and provide a learning-friendly codebase with conventional patterns.

Objective

- Primary Objective: Implement a full small e■commerce workflow from product listing to order completion.
- Secondary Objectives:
- Provide clean, documented code suitable for learning and demonstration.
- Ensure cart merging logic when a guest logs in.
- Provide admin UX improvements (thumbnails, inlines, filters).
- Keep code modular (separate apps): `accounts`, `store`, `carts`, `orders`, `category`.

System / Process Design

- High-level Architecture:
- Presentation: Django templates + Bootstrap 5 + jQuery.
- Backend: Django apps per domain area (MVT).
- Persistence: Django ORM (SQLite for development).
- Components:
- `accounts` app: Custom `Account` user model, `UserProfile`.
- `store` app: `Product`, `Variation`, `ReviewRating`, `ProductGallery`.
- `carts` app: `Cart`, `CartItem` and session management.
- `orders` app: `Order`, `OrderProduct`, payment handling.
- `category` app: product categorization and navigation context processor.
- Data flow (simplified):
- User browses `store` → adds product/variation to `cart` → when ready, `checkout` → `place_order` creates `Order` and `OrderProduct` records → payments endpoint updates `Payment` and mark `Order` as placed.

Tools / Technology Used

- Language & Framework: Python 3.x, Django 3.2.x (project uses Django MVT).
- Packages: Pillow (image fields), python-decouple (env config), requests (optional), other default Django packages. See `requirements.txt`.
- Frontend: Bootstrap 5 (CSS), jQuery 2.x, Font Awesome icons, Roboto fonts.

- DB: SQLite3 (dev). Recommend PostgreSQL for production.
- Dev & Ops: `manage.py` for Django commands, local `media/` and `static/` for assets. No CI/CD included in codebase.

Flowchart / Algorithm Modules

- Main Flows:
- Product browsing → filter by category → view product detail.
- Add to cart → (session cart) merge with user cart on login → update cart quantities → checkout.
- Place order → create Order + OrderProduct entries → payment confirmation → mark items ordered.
- Key Algorithms / Logic:
- Cart merge algorithm: match cart items by product + variation set when user logs in; merge quantities and delete session cart.
- Price calculation: apply product base price, variation price modifiers, multiply by quantity, apply taxes/discounts (if implemented).
- Review submission: check if user purchased product before allowing review (if enforced).
- Suggested Flowchart Steps:
- Start → Load home/store.
- User selects product and variations → Add to cart.
- If guest: store cart in session; if auth: store cart in DB linked to user.
- On login: attempt cart merge (match by product+variations) → update DB cart.
- User proceeds to checkout → validates shipping/payment → place order.
- Payment endpoint processes/records payment → update order status → notify user.
- End.

Description of modules

- `accounts`:
- Responsibilities: custom `Account` model (email login), registration, login/logout, email activation, password reset, profile edit, user dashboard and order history.
- Key files: `accounts/models.py`, `accounts/forms.py`, `accounts/views.py`, `accounts/urls.py`, `accounts/admin.py`.
- `store`:
- Responsibilities: product CRUD (admin), product listing, category filtering, product detail with variations and gallery, reviews.
- Key files: `store/models.py`, `store/views.py`, `store/forms.py`, `store/admin.py`, `store/urls.py`.
- `carts`:
- Responsibilities: manage cart/session, add/remove items, variation handling, provide `counter` context processor for cart count.
- Key files: `carts/models.py`, `carts/views.py`, `carts/context_processors.py`, `carts/urls.py`.
- `orders`:
- Responsibilities: checkout flow, order creation, order-product relationships, payment recording, admin order management.
- Key files: `orders/models.py`, `orders/views.py`, `orders/forms.py`, `orders/admin.py`, `orders/urls.py`.
- `category`:
- Responsibilities: category model and menu links context processor used by templates to build navigation.
- Key files: `category/models.py`, `category/context_processors.py`, `category/admin.py`.
- `greatkart` (project):
- Responsibilities: global settings, URL includes, static/media setup, global templates.
- Key files: `greatkart/settings.py`, `greatkart/urls.py`.

Output analysis

- Primary Outputs:
- Web UI pages: homepage, store listing, product detail, cart, checkout, order confirmation.
- Admin interfaces: product listing, inline gallery editing, order management.
- Database records: Products, Variations, Cart/CartItems, Orders, Payments.
- Metrics to evaluate (for testing/demo):
- Functional correctness: product listing, add/remove cart items, place order.
- Cart merge accuracy: percentage of items merged versus duplicated.
- Admin productivity: time to add/edit product with galleries (qualitative).
- Error rates: failed checkout flows, validation problems.
- Test cases to run:
- Add multiple variation combinations to cart as guest, log in, verify merge.
- Submit review as user who purchased the product.
- Create product with multiple gallery images and ensure admin thumbnails render.

Summary of Finding

- What works well: Modular app separation (accounts, store, carts, orders, category) maps nicely to real-world e■commerce domains; admin enhancements (thumbnails, inlines) improve management; cart merge logic addresses a common practical problem; templates and static assets provide a presentable demo UI.
- Gaps / Recommendations:
- Replace placeholder payment flow with real gateway for production.
- Add rate-limited email sending for verification and consider async tasks (Celery) for heavy jobs.
- Add integration tests covering cart merge and checkout flows.
- Update `TECH_STACK.md` or `SYNOPSIS.md` to include admin customization details and a full URL reference table (optional but useful for HOD demo).
- Conclusion: The project is a complete, well-structured Django learning/demo application covering the essential e■commerce workflow; it’s ready for presentation with minor enhancements suggested above.