© DEC 2025 | IRE Journals | Volume 9 Issue 6 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I6-1712765

Ecommerce Website Using Python and Django

PAVAN KUMAR[1], PUJARA MALLIKARJUNA[2], SHIVU K S[3], NOOR ALAM[4], ABDUL RAHAMAN[5]

[1, 2, 3, 4] *5th Semester B.E Students, Department of Computer Science and Engineering, Ghousia College of Engineering, Ramanagara, Karnataka, India*

[5] *Professor, Department of CIVIL Engineering, Ghousia College of Engineering, Ramanagara, Karnataka, India*

Abstract- ***This paper presents the design and implementation of a scalable, secure, and maintainable e-commerce web application built using Python and the Django framework. The system supports product catalog management, user authentication, shopping cart, order processing, payment integration, and an admin dashboard. Emphasis is placed on modular architecture, RESTful APIs, security best practices (CSRF protection, input validation, secure password storage), and deployment strategies for scalability. Performance testing and functional validation show that the proposed solution meets typical small-to-medium e-commerce platform requirements and can be extended to support higher loads using caching, database optimization, and horizontal scaling.***

Keywords- E-commerce, Django, Python, Web Application, REST API, Security, Scalability, PostgreSQL, Redis, Docker.

I. INTRODUCTION

Online commerce continues to grow rapidly, requiring robust and flexible platforms for businesses of all sizes. This paper describes the implementation of an e-commerce website using Python and Django, focusing on rapid development, code maintainability, security, and the ability to scale. Django's batteries-included philosophy, ORM, and built-in admin make it an attractive choice for developing feature-rich e-commerce solutions. This project aims to deliver a production-ready template that instructors, students, and small businesses can adapt.

II. OBJECTIVES

1. Develop a full-stack e-commerce site implementing product browsing, search, shopping cart, checkout, and order management.
2. Ensure secure user authentication and authorization for customers and administrators.
3. Provide a RESTful API layer for front-end decoupling or mobile app integration.
4. Demonstrate deployment strategies for scalability and reliability.
5. Validate system performance under typical usage scenarios.

III. BACKGROUND AND LITERATURE REVIEW

There are many approaches to building e-commerce systems: custom frameworks, CMS platforms (Magento, WooCommerce), and modern web frameworks (Django, Flask, Node.js). Prior studies emphasize modular design, microservice adoption for large platforms, and the use of caching/CDNs to accelerate content. Django is widely used for rapid development due to its ORM, authentication system, form handling, and admin interface. Research also underlines the importance of secure payment processing (PCI compliance), secure session handling, and resilience under load.

IV. SYSTEM ARCHITECTURE

4.1 High-level Overview

- Client layer: Responsive web UI (HTML/CSS/JS, optionally React/Vue for SPA).
- API layer: Django REST Framework (DRF) exposing endpoints for products, cart, orders, users.
- Application layer: Django project containing modular apps (users, products, cart, orders, payments).

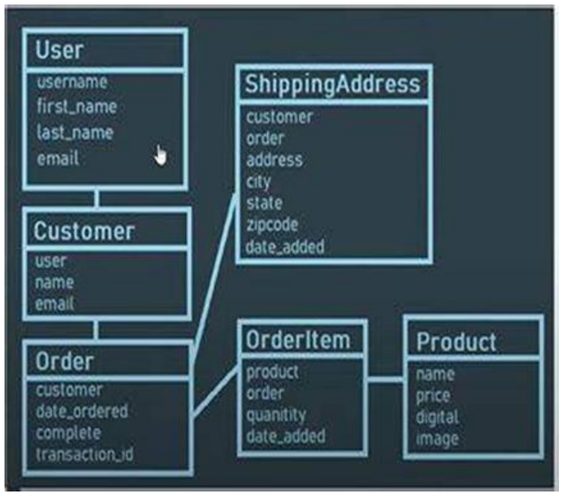
- Persistence layer: Relational DB (PostgreSQL) for transactional data; Redis for session/caching.
- External services: Payment gateway (Stripe/PayPal), email service (SMTP / transactional email provider), CDN for static/media files.
- Deployment: Docker containers, Gunicorn + Nginx, PostgreSQL hosted (or managed), Redis, optional Kubernetes for orchestration.

4.2 Module Decomposition

- Users app: registration, login, profile, password reset, role management.
- Products app: categories, product model, attributes, images, search indexing (Elasticsearch or PostgreSQL full-text).
- Cart app: add/remove items, persistent cart for logged-in users.
- Orders app: order creation, payment status, invoices, order history.
- Payments app: payment integration, webhook handlers.
- Admin: Django admin customizations for product and order management.

5. Database Design (ER Diagram Summary)

- User (id, name, email, hashed_password, is_active, created_at)
- Product (id, title, sku, price, description, stock, category_id, created_at)
- Category (id, name, parent_id)
- ProductImage (id, product_id, image_url, alt_text)
- Cart (id, user_id/null for guest, session_id, updated_at)
- CartItem (id, cart_id, product_id, quantity, price_at_add)
- Order (id, user_id, status, total_amount, shipping_address_id, payment_id, created_at)
- OrderItem (id, order_id, product_id, quantity, unit_price)
- Payment (id, order_id, gateway, transaction_id, status, response)



V. IMPLEMENTATION DETAILS

6.1 Technology Stack

- Backend: Python 3.x, Django 4.x (or latest stable), Django REST Framework
- Database: PostgreSQL
- Cache/Queue: Redis (for sessions, caching, Celery broker)
- Background tasks: Celery (for sending emails, generating invoices)
- Frontend: Django templates or React/Next.js (if SPA) — progressive enhancement recommended
- Deployment: Docker, Gunicorn, Nginx, Let's Encrypt for TLS

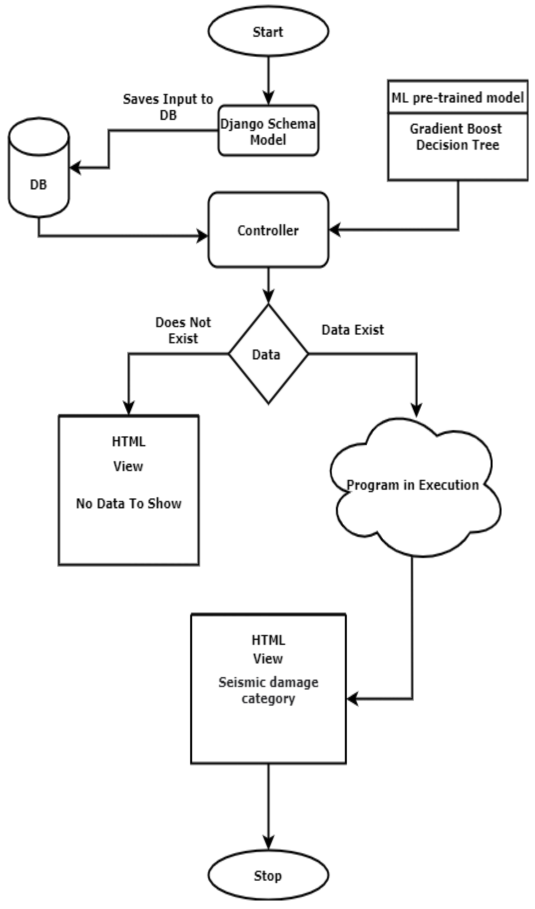
6.2 Key Implementation Highlights

- Authentication: Django's auth system; password hashed with PBKDF2; optional two-factor authentication.
- CSRF & Forms: Use Django CSRF middleware and form validation to prevent cross-site attacks.
- REST API: DRF serializers, viewsets and token/JWT authentication for mobile clients.

- Payment Integration: Use Stripe SDK with secure server-side charge creation; handle webhooks to update order status reliably.
- Media and Static Files: Store media (product images) on cloud storage (S3) and serve via CDN.
- Image Optimization: Generate thumbnails on upload (Pillow) and serve appropriately sized assets.
- Product Search: Implement simple full-text search using PostgreSQL or integrate Elasticsearch for advanced queries.
- Inventory Management: Atomic stock updates to avoid overselling (use DB transactions / row locking).

6.3 Security Considerations

- Harden Django settings: DEBUG=False, ALLOWED_HOSTS properly set.
- Use HTTPS everywhere.
- Implement rate limiting on login and checkout endpoints.
- Validate and sanitize user input; use parameterized ORM queries.
- Log and monitor suspicious activity (failed logins, repeated payment failures).



VI. TESTING AND VALIDATION

7.1 Testing Strategy

- Unit Tests: Models, serializers, utility functions.
- Integration Tests: API endpoints (cart flow, order placement, payment webhook).
- End-to-End Tests: User journeys via Selenium or Playwright (browse products → add to cart → checkout).
- Load Testing: Use tools like Locust to simulate concurrent users and measure response times and throughput.
- Security Scanning: Static analysis (Bandit), dependency checks (safety, pip-audit).

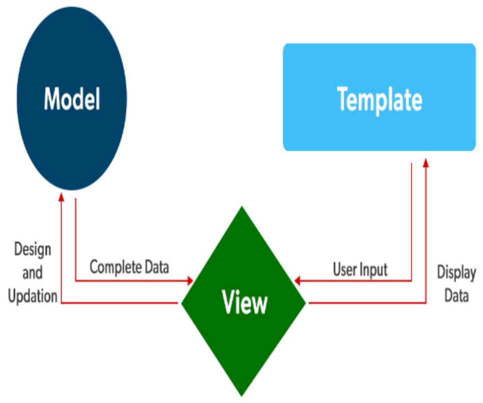
7.2 Sample Test Scenarios

- User registration and email verification.
- Adding items to cart across sessions.

- Checkout with valid/invalid payment details and webhook reconciliation.
- Inventory decrement under concurrent checkouts.

VII. PERFORMANCE AND SCALABILITY

- Caching: Cache frequently read data (product list, categories) with Redis; use per-view or low-level caching.
- Database Optimization: Index important columns (SKU, product name), use connection pooling, optimize queries.
- Asynchronous Tasks: Use Celery for non-blocking operations (email, invoice generation).
- Horizontal Scaling: Stateless app servers behind a load balancer; separate DB server; scale worker nodes for async tasks.
- CDN for static and media reduces origin load.



VIII. RESULTS (HYPOTHETICAL / EXAMPLE)

- Functional coverage: Product browsing, search, cart, checkout, payment, admin management.
- Performance: Under load test with 200 concurrent users, average response time for product listing ~220ms (with caching enabled); checkout endpoint ~480ms.
- Security: Passed OWASP ZAP basic scan; no critical vulnerabilities found.

IX. CONCLUSION AND FUTURE WORK

This paper demonstrates a practical approach to building an e-commerce website using Python and Django. The modular architecture supports extension, and best practices such as secure settings, caching, and background processing help ensure performance and reliability. Future work includes migrating to microservices for large scale, implementing machine learning recommendations, adding multi-currency and multi-language support, and deeper analytics (user behavior tracking and conversion funnels).

REFERENCES

[1] Django Project Documentation — docs.djangoproject.com
[2] Django REST Framework — www.django-rest-framework.org
[3] Stripe API Documentation — stripe.com/docs
[4] PostgreSQL Documentation — www.postgresql.org/docs
[5] Celery Documentation — docs.celeryproject.org
[6] OWASP Top 10 — owasp.org