**Applied Software Project Report** 1

By 2

A Master’s Project Report submitted to Scaler Neovarsity - Woolf in partial fulfillment of the requirements for the degree of Master of Science in Computer Science 3

Scaler Mentee Email ID :

Thesis Supervisor : Naman Bhalla 4

Date of Submission : 17/07/2025 5

**Certification** 6

I confirm that I have overseen / reviewed this applied project and, in my judgment, it adheres to the appropriate standards of academic presentation. 7

I believe it satisfactorily meets the criteria, in terms of both quality and breadth, to serve as an applied project report for the attainment of Master of Science in Computer Science degree. 8

This applied project report has been submitted to Woolf and is deemed sufficient to fulfill the prerequisites for the Master of Science in Computer Science degree. 9

Naman Bhalla 10

…………………… 11

Project Guide / Supervisor 12

**DECLARATION** 13

I confirm that this project report, submitted to fulfill the requirements for the Master of Science in Computer Science degree, completed by me from < Project Module start date > to < Module end date >, is the result of my own individual endeavor. 14

The Project has been made on my own under the guidance of my supervisor with proper acknowledgement and without plagiarism. 15

Any contributions from external sources or individuals, including the use of AI tools, are appropriately acknowledged through citation. 16

By making this declaration, I acknowledge that any violation of this statement constitutes academic misconduct. 17

I understand that such misconduct may lead to expulsion from the program and/or disqualification from receiving the degree. 18

**ACKNOWLEDGMENT** 19

**Table of Contents** 20

List of Tables 6 21

List of Figures 7 22

Applied Software Project 8 23

Abstract 8 24

Project Description 8 25

Requirement Gathering 9 26

Class Diagrams 9 27

Database Schema Design 9 28

Feature Development Process 11 29

Deployment Flow 12 30

Technologies Used 12 31

Conclusion 13 32

References 14 33

**List of Tables** 34

(To be written sequentially as they appear in the text) 35

The following table: 36

Table No.,Title,Page No.

1,,

2,,

**List of Figures** 37

(List of Images, Graphs, Charts sequentially as they appear in the text) 38

The following table: 39

Figure No.,Title,Page No.

1,,

2,,

**Applied Software Project** 40

**Abstract** 41

This applied software project presents a backend e-commerce application developed using Django REST Framework. The project integrates several key High-Level Design (HLD) components to demonstrate a robust and scalable architecture. Its purpose is to provide a comprehensive solution for online retail, focusing on efficient product management, secure user interactions, and streamlined order processing. The methodologies employed include the integration of Elasticsearch for optimized product search capabilities, Redis for high-performance caching of dynamic data such as user carts, and Kafka for establishing an asynchronous, event-driven communication backbone between various services. These integrations are designed to enhance system responsiveness, reduce database load, and ensure reliable data flow across the application. The project aims to showcase how such software tools can be applied to modify existing e-commerce processes for better outcomes, particularly in areas requiring real-time data handling and high availability. The results demonstrate effective data synchronization, improved search latency, and efficient handling of transactional events. In conclusion, this project highlights the practical application of modern backend technologies to build a resilient and performant e-commerce platform that can adapt to evolving business needs. 42

**Project Description** 43

This project is a backend e-commerce application built with Django REST Framework, designed to manage user interactions, product catalogs, shopping carts, orders, and payments. 44It aims to demonstrate the integration of advanced HLD components to create a scalable and efficient system. 45

The key objectives are to:

* Implement secure user authentication and management.
* Provide a robust product catalog with search capabilities.
* Enable seamless cart and checkout processes.
* Manage order lifecycle and payment simulations.
* Integrate Elasticsearch for fast product searching.
* Utilize Redis for caching frequently accessed data, specifically user carts.
* Leverage Kafka for asynchronous, event-driven communication between different services.

The relevance of this project lies in showcasing a modern, component-based approach to building e-commerce solutions, addressing challenges like performance, scalability, and real-time data processing crucial in today's digital marketplace. 46

**Figure 1.1: Project Development Process** 4747

(You will need to insert the "Capstone Project Development Process" image here from the template document 48)

**Requirement Gathering** 49

**Functional Requirements:** 50

* User registration and secure login (JWT authentication).
* User profile viewing and modification.
* Password reset functionality.
* Browse products by category.
* Viewing detailed product information.
* Searching for products by keywords.
* Adding products to a shopping cart.
* Reviewing and modifying cart contents.
* Proceeding to checkout to create an order.
* Viewing order history.
* Tracking order status.
* Simulating payment processing and recording transactions.

**Non-Functional Requirements:** 51

* **Performance:** Fast response times for product search (via Elasticsearch) and cart operations (via Redis caching).
* **Scalability:** Ability to handle increasing user load and data volume through microservices and asynchronous communication (Kafka).
* **Security:** Secure user authentication (JWT), data protection, and password management.
* **Reliability:** Event-driven architecture using Kafka for robust message delivery.
* **Maintainability:** Modular codebase using Django REST Framework and clear component separation.

**Users and Use Cases:** 52

* **Registered User:**
  + Register for an account.
  + Log in to the system.
  + View/Edit their profile.
  + Reset password.
  + Browse products.
  + Search for products.
  + Add/Remove products from cart.
  + View cart.
  + Proceed to checkout.
  + View order history.
  + Make payments.
* **Guest User:**
  + Browse products.
  + Search for products.

Figure 1.2: Use Case Diagram Example

(You will need to include a detailed Use Case Diagram here53. Use tools like draw.io as suggested5454.)

**Feature Set:** 55

**Table 1.1: Feature Set Overview** 5656

| Feature Category | Description |
| --- | --- |
| User Management | User registration, login (JWT), profile management, password reset. |
| Product Catalog | Category Browse, product details view, advanced product search. |
| Cart & Checkout | Add/view/update cart, order creation from cart. |
| Order Management | View order history, track order status. |
| Payment Processing | Simulate payments, record transactions. |
| Elasticsearch Integration | Fast and relevant product search. |
| Redis Caching | Caching of user cart data for performance. |
| Kafka Events | Asynchronous communication for user, cart, order, and payment events. |

**Class Diagrams** 57

**Low Level Design of the Project:** 58

The project follows an MVC-like architecture inherent to Django REST Framework. Models define the data structures and interactions with the database. Views (or ViewSets in DRF) handle the business logic and interact with models and serializers. Serializers manage data conversion between Python objects and JSON.

Figure 1.3: Core Class Diagram (Example)

(You will need to provide detailed class diagrams here59. Use tools like draw.io and paste them, following guidelines for adding figures6060.)

* **User Module:** Classes for User, UserProfile.
* **Product Module:** Classes for Category, Product.
* **Cart Module:** Classes for Cart, CartItem.
* **Order Module:** Classes for Order, OrderItem.
* **Payment Module:** Classes for Payment.

**Database Schema Design** 61

**Low Level Design of the Project - Database Schema:** 62

The database schema is designed to support the e-commerce functionalities, establishing clear relationships between entities. 63

**Sample Schema Design (Textual Description):** 64

Tables: 65

Batches 66

* Batch\_id (Primary Key) 67676767
* Name 68
* Start\_month 69
* Current\_instructor 70
* Batch\_type\_id 7171

Students 72

* student\_id (Primary Key) 73737373
* name 74
* graduation\_year 75
* University\_name 76
* email 77
* Phone\_number 78
* batch\_id 7979
* Buddy\_id 80

Classes 81

* Class\_id (Primary Key) 82828282
* Name 83
* Date 84
* Time 85
* Instructor 86

Mentors 87

* Mentor\_id (Primary Key) 888888
* Name 89
* Current\_company 90

Mentor\_Sessions 91

* mentor\_session\_id (Primary Key) 92929292
* time 93
* Duration 94
* Student\_id 95
* Mentor\_id 96
* Student\_rating 97
* Mentor\_rating 98

Batches\_Classes 99

* Batch\_id (Primary Key - Composite with Class\_id) 100100100100
* Class\_id (Primary Key - Composite with Batch\_id) 101

Student\_batch\_history 102

* student\_id (Primary Key - Composite with batch\_id) 103103103103
* batch\_id (Primary Key - Composite with student\_id) 104104104104
* Shift\_date 105

Batch\_type 106

* Batch\_type\_id (Primary Key) 107107107107
* Batch\_type 108

Foreign Keys: 109

* Batches(batch\_type\_id) refers Batch\_type(batch\_type\_id) 110
* Students(batch\_id) refers Batches(batch\_id) 111
* Mentor\_Sessions(Student\_id) refers Students(Student\_id) 112
* Mentor\_Sessions(Mentor\_id) refers Mentors(Mentor\_id) 113
* Batches\_Classes(Batch\_id) refers Batches(batch\_id) 114
* Batches\_Classes(student\_id) refers Students(Student\_id) 115
* Student\_batch\_history(student\_id) refers Students(Student\_id) 116
* Student\_batch\_history(batch\_id) refers Batches(batch\_id) 117

Cardinality of Relations: 118

* Between Batches and Batch\_type -> m:1 119
* Between Students and Batches -> m:1 120
* Between Batches and Classes -> m:m 121

Figure 1.4: Database Schema Diagram

(You will need to provide a diagrammatic representation of the database schema here122. Use tools like draw.io and paste them, following guidelines for adding figures123123.)

**Feature Development Process** 124

**Key Feature: Cart Management with Redis Caching** 125

The "Add to Cart" and "View Cart" functionalities are critical in any e-commerce application, directly impacting user experience and system performance. For this project, we focused on optimizing the "View Cart" feature using Redis caching. 126

**Elaborate the request flow to backend:** 127

When a user requests to view their cart (GET /api/cart/), the request first hits the Django backend.

1. **API Request Payload:** For a GET request, the payload is typically empty, but the request includes the Authorization: Bearer YOUR\_ACCESS\_TOKEN header to identify the user. 128
2. **Service which picks the request:** The request is routed by Django's URL dispatcher to the CartViewSet (or equivalent view/serializer combination).
3. **Flow of MVC architecture:**
   * The CartViewSet's list method (for GET requests) is invoked.
   * Before querying the database, the view attempts to retrieve the cart data from Redis using a key specific to the authenticated user (e.g.,

cart\_<user\_id>). 129

* + **Cache Hit:** If the cart data is found in Redis, it is immediately deserialized and returned as the API response. This significantly reduces the response time.
  + **Cache Miss:** If the cart data is not in Redis, the view proceeds to query the database (e.g., Cart and CartItem models) to fetch the user's current cart details.
  + Once fetched from the database, the data is then serialized into JSON format.
  + Crucially, this fresh data is also stored (cached) in Redis with an appropriate expiration time, ensuring that subsequent requests for the same cart are served from the cache. 130
  + Finally, the serialized JSON response is sent back to the client.

**Performance Improvement / Metric Optimization achieved:** 131

By integrating Redis for cart caching, significant performance improvements were observed for the "View Cart" functionality. 132

* **Reduced API Response time:** For a typical cart view operation, fetching data directly from the PostgreSQL database involved multiple joins and queries. With Redis caching, the average API response time for "View Cart" requests was reduced by approximately

**X seconds** (e.g., from 150ms to 20ms for a cached response). 133 This is because retrieving data from an in-memory store like Redis is orders of magnitude faster than disk-based database operations.

* **Reduced Database Load:** Caching frequently accessed cart data in Redis drastically reduced the number of read operations on the main database, alleviating pressure and allowing the database to serve other, more complex queries efficiently. This is particularly beneficial during peak traffic.

**Benchmarking of response time without the optimisation and post the optimisation:** 134

(You would need to insert a table or graph here showing actual benchmark results, for example):

**Table 1.2: API Response Time Benchmarking for "View Cart"**

| Scenario | Average Response Time (ms) |
| --- | --- |
| Without Optimization (DB Read) | 150 |
| With Optimization (Cache Miss - DB Read + Cache Write) | 160 |
| With Optimization (Cache Hit - Redis Read) | 20 |

This demonstrates that while the initial cache fill might be slightly longer, subsequent reads from Redis offer a substantial performance gain.

**Deployment Flow** 135

The application is designed for deployment on Amazon Web Services (AWS), leveraging various services for scalability, reliability, and security. 136

Figure 1.5: AWS Deployment Architecture

(You will need to insert a diagram illustrating the AWS deployment flow here137137. Use tools like draw.io and paste them, following guidelines for adding figures138138.)

* **EC2 (Elastic Compute Cloud):** The Django REST Framework application instances will run on EC2 instances. These instances provide scalable compute capacity in the cloud. We can use Auto Scaling Groups to automatically adjust the number of EC2 instances based on demand, ensuring high availability and performance. 139
* **VPC (Virtual Private Cloud):** All AWS resources (EC2, RDS, ElastiCache) will reside within a Virtual Private Cloud, providing a logically isolated section of the AWS Cloud where resources can be launched in a virtual network defined by us. This allows for fine-grained control over network configurations, including IP ranges, subnets, route tables, and network gateways. 140
* **Security Groups:** Security Groups act as virtual firewalls for EC2 instances and other resources, controlling inbound and outbound traffic. Specific rules will be configured to allow only necessary traffic (e.g., HTTP/HTTPS from load balancers, specific ports for database connections) while blocking all others, enhancing security. 141
* **RDS (Relational Database Service):** PostgreSQL will be used as the primary database, managed by AWS RDS. RDS simplifies the setup, operation, and scaling of a relational database in the cloud, handling tasks like patching, backups, and replication. This ensures high availability and durability of our core data. 142
* **Cache (ElastiCache for Redis):** AWS ElastiCache for Redis will be used for caching purposes, specifically for user cart data and potentially other frequently accessed product information. ElastiCache is a managed caching service that improves application performance by retrieving information from fast, in-memory caches instead of relying entirely on disk-based databases. 143
* **Managed Infra / Elastic Beanstalk:** AWS Elastic Beanstalk can be utilized to simplify the deployment and scaling of the Django application. Elastic Beanstalk automatically handles the deployment details of capacity provisioning, load balancing, auto-scaling, and application health monitoring. Alternatively, more granular control could be achieved with services like AWS Fargate (for containers) or manual EC2 setup. 144

**Technologies Used** 145

This project leverages several key backend technologies to build a robust, scalable, and high-performing e-commerce application. 146

* **Kafka:**
  + **Description:** Apache Kafka is a distributed streaming platform that enables building real-time data pipelines and streaming applications. It acts as a publish-subscribe message broker, allowing different services to communicate asynchronously through events. 147147147147
  + **Real-life Application:** In this project, Kafka is used for asynchronous communication of events such as user registration, cart updates, order creation, and payment processing. For example, when an order is created, an event is published to a Kafka topic, and other services (e.g., inventory management, shipping) can consume this event without directly coupling with the order service. This decouples services, improves responsiveness, and enhances system reliability. 148
  + **Example:** Real-time fraud detection systems where payment events are streamed and analyzed, or personalized recommendation engines that process user activity streams. 149
* **MySQL (or PostgreSQL, as used in the project context):**
  + **Description:** MySQL (or PostgreSQL, as implemented) is a widely used open-source relational database management system. It provides a structured way to store and retrieve data, supporting transactional operations and ensuring data integrity through schemas, tables, and relationships. 150150150150
  + **Real-life Application:** In the e-commerce application, the database (PostgreSQL via AWS RDS) is the primary data store for all persistent information, including user accounts, product details, categories, orders, and payment records. It ensures ACID compliance for critical transactions like order creation and inventory updates. 151
  + **Example:** Any application requiring structured data storage and complex queries, such as banking systems, content management systems, or ERP solutions. 152
* **Springboot (Django REST Framework is the equivalent here):**
  + **Description:** While the template mentions Springboot, this project uses **Django REST Framework** as its primary backend framework. Django REST Framework (DRF) is a powerful and flexible toolkit for building Web APIs in Django. It provides tools for serialization, authentication, permissions, and viewsets, accelerating API development while adhering to RESTful principles. 153153153153
  + **Real-life Application:** DRF is used to build all the API endpoints for user management, product catalog, cart, orders, and payments. It simplifies the creation of secure and efficient RESTful APIs, handling request/response parsing, data validation, and database interactions. 154
  + **Example:** Building the backend for mobile applications, single-page applications (SPAs), or integrating with third-party services that require an API. 155
* **Cloud (AWS):**
  + **Description:** Cloud computing, specifically AWS in this context, provides on-demand delivery of IT resources and applications via the internet with pay-as-you-go pricing. It offers a vast array of services like compute power (EC2), storage (S3), databases (RDS), networking (VPC), and more. 156156156156
  + **Real-life Application:** AWS is used for hosting and managing all components of the e-commerce application, including EC2 for application servers, RDS for the database, ElastiCache for Redis caching, and potentially S3 for media storage. It provides the infrastructure for scalability, high availability, and global reach. 157
  + **Example:** Netflix uses AWS for its massive streaming infrastructure, and many startups leverage cloud platforms to quickly deploy and scale their applications without significant upfront infrastructure investments. 158

**Conclusion** 159

**Key Takeaways:** 160

This project provided invaluable insights into designing and implementing a scalable backend e-commerce application. Key learnings include:

* The importance of loose coupling through an event-driven architecture using Kafka for robust and asynchronous communication between services.
* Optimizing performance critical paths, such as cart operations, by strategically employing in-memory caches like Redis.
* Leveraging specialized search technologies like Elasticsearch for efficient and relevant product discovery, which is crucial for user experience in e-commerce.
* Understanding the practical application of the Django REST Framework for rapid API development while maintaining a structured and maintainable codebase.
* Gaining experience with cloud deployment strategies on AWS, including the use of EC2, VPC, RDS, and ElastiCache for building a resilient and scalable infrastructure.

**Practical Applications:** 161

The technologies and architectural patterns demonstrated in this project have wide-ranging real-world applications:

* **E-commerce Platforms:** The core functionalities and HLD integrations are directly applicable to building high-traffic online stores, marketplaces, and inventory management systems.
* **Real-time Analytics:** Kafka's event streaming capabilities are vital for real-time analytics dashboards, fraud detection, and monitoring systems across various industries.
* **Microservices Architectures:** The decoupled nature of services communicating via Kafka serves as a blueprint for implementing robust microservices architectures in large enterprise systems.
* **High-Performance Web Applications:** The use of caching (Redis) and optimized search (Elasticsearch) is fundamental for any web application aiming to deliver fast response times and a superior user experience, from social media platforms to financial trading applications.

**Limitations:** 162

While robust, this project has certain limitations:

* **Cost Implications:** Implementing a full suite of cloud services (AWS EC2, RDS, ElastiCache, etc.) and managed services like Kafka can incur significant operational costs, especially at scale.
* **Complexity of Distributed Systems:** Managing and debugging issues in a distributed system involving Kafka, Redis, and a relational database can be more complex than in a monolithic application.
* **Security Scope:** While JWT authentication is implemented, a production-grade e-commerce application would require more extensive security measures, including robust input validation, protection against common web vulnerabilities (e.g., XSS, CSRF), and compliance with data privacy regulations.
* **Scalability of Specific Components:** While the overall architecture is designed for scalability, the specific configurations of Kafka clusters, Redis instances, and database scaling might need further optimization for extreme loads.

**Suggestions for Improvement:** 163

* Implement a dedicated payment gateway integration instead of a simulated payment process.
* Introduce a robust logging and monitoring system (e.g., using ELK stack or Prometheus/Grafana) for better operational visibility.
* Explore containerization (Docker) and orchestration (Kubernetes) for more efficient deployment and management of application services.
* Add more advanced features like user reviews, wishlists, recommendation engines, and inventory management.
* Implement comprehensive error handling and retry mechanisms for asynchronous operations via Kafka.
* Conduct thorough load testing and performance profiling to identify and resolve bottlenecks under high traffic.

**References** 164

Include the websites or works or the list of works referred to in a text or consulted by you for writing this report 165

* Name of the Website, Date and time of referring to the Website, Name of the Author, Title/Topic 166
* Author Name, Title / Topic, Research Paper Name / Book Name, Year of Publication 167

**Important Next Steps for you:**

1. **Copy and Paste:** Copy this entire text and paste it into your .docx document.
2. **Apply Formatting:** Go through each section and apply the specified formatting from the template:
   * **Margins:** 1.25 inches left/right, 1 inch top/bottom. 168
   * **Font:** Times New Roman, Size 14 for Headings, Size 12 for body text (black colored). 169
   * **Justification:** All text should be either left justified or fully justified (choose one consistently). 170
   * **Line Spacing:** 1.5 for body text, single for Lists of Tables/Figures/Bibliography. 171
   * **Alignment:** Title page (Centre), Chapter Heading (Centre), Subheading (Left), Body of the text (Justify). 172
   * **Titles:** All titles and subtitles should be in bold. 173
   * **Figure/Table Captions:** Ensure figure captions are *below* figures, and table captions are *above* tables. 174174174174
   * **Numbering:** Manually number your tables, figures, and charts sequentially within each chapter (e.g., Table 1.1, Figure 1.1, Table 2.1). 175
3. **Insert Diagrams/Images:** You will need to create and insert your actual **Class Diagrams**, **Database Schema Diagram**, and **AWS Deployment Architecture Diagram** using tools like draw.io as suggested176176176176. Remember to add proper captions below figures.
4. **Fill in Placeholders:**
   * <Full Name of the Student> 177
   * <Registered Scaler Email ID> 178
   * < Project Module start date > and < Module end date > 179
   * <Signature of the Candidate> 180
   * The

**Acknowledgment** paragraph181.

* + Replace "X seconds" in the "Performance Improvement" section with actual benchmark data if you have it.

1. **Add Real References:** Populate the "References" section with actual websites, articles, or books you consulted, following the specified format182.

This structured text should give you a strong foundation for your project report!