Development of a Lightweight, Secure, and Responsive E-commerce Platform Using Next.js and Express.js

A Thesis Submitted to

FACULTY OF ENGINEERING AND TECHNOLOGY
JADAVPUR UNIVERSITY

Submitted in partial fulfillment of the requirements for the degree of,
MASTER OF COMPUTER APPLICATION

By
Dipendu Dhali

Examination Roll No.: MCA00254026
Class Roll No.: 002310503026
Registration No: 1661882 of 2023 – 2024

Under the guidance of
Prof. Subhadip Basu

Jadavpur University
188, Raja S.C.Mallick Rd,
Kolkata – 700032, West Bengal, India

CERTIFICATE OF RECOMMENDATION

This is to certify that the work embodied in this thesis entitled **"Development of a Lightweight, Secure, and Responsive E-commerce Platform Using Next.js and Express.js"** has been satisfactorily completed by Dipendu Dhali (Registration Number 1661882 of 2023 - 2024; Class Roll No. 002310503026; Examination Roll No. MCA00254026). It is a bona fide piece of work carried out under my supervision and guidance at Jadavpur University, Kolkata, for partial fulfillment of the requirements for the awarding of the Master of Computer Application degree of the Department of Computer Science and Engineering, Faculty of Engineering and Technology, Jadavpur University, during the academic year 2023 - 2025.

Prof. (Dr.) Nirmalya Chowdhury,
Head of the Department,
Department of Computer Science and Engineering,
Jadavpur University

Prof. (Dr.) Subhadip Basu,
Department of Computer Science and Engineering,
Jadavpur University
(Supervisor)

Prof. (Dr.) Saroj Mandal,
DEAN, Faculty of Engineering and Technology,
Jadavpur University

CERTIFICATE OF APPROVAL

This is to certify that the project report entitled **"Development of a Lightweight, Secure, and Responsive E-commerce Platform Using Next.js and Express.js"** is a bonafide record of work carried out by **Dipendu Dhali**, in fulfillment of the requirements for the award of the degree of **Master of Computer Application** in the **Department of Computer Science and Engineering, Jadavpur University**. It is understood that by this approval the undersigned do not necessarily endorse or approve any statement made, opinion expressed or conclusion drawn therein but approve the project report only for the purpose for which it has been submitted.

_____

Signature of Examiner 1
Date:

_____

Signature of Examiner 2
Date:

DECLARATION OF ORIGINALITY AND COMPLIANCE OF ACADEMIC ETHIC

I hereby declare that the thesis entitled **"Development of a Lightweight, Secure, and Responsive E-commerce Platform Using Next.js and Express.js"** contains literature survey and original research work by the undersigned candidate, as a part of his degree of Master of Computer Application, Jadavpur University. All the information has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all materials and results that are not original to this work.

Name: Dipendu Dhali

Examination Roll No.: MCA00254026

Class Roll No.: 002310503026

Registration No.: 1661882 of 2023 – 2024

Thesis Title: Development of a Lightweight, Secure, and Responsive E-commerce Platform Using Next.js and Express.js

Signature of the Candidate:

Dipendu Dhali

Acknowledgment

I would like to express my heartfelt gratitude to my supervisor, **Prof. Subhadip Basu**, for providing me with the opportunity to work under his guidance for the successful completion of my project. His continuous support, encouragement, and valuable insights have been instrumental throughout this journey.

From the very beginning, he has shown great patience and understanding, always encouraging me to think independently and explore new ideas. His constructive feedback and timely suggestions helped me overcome many challenges during the development process and enriched my learning experience.

I am especially thankful to him for granting me the academic freedom to pursue my interests in web development, which allowed me to dive deep into this domain with confidence and creativity. His trust in my abilities and consistent motivation helped me stay focused and committed to my goals.

Working under his supervision has not only enhanced my technical skills but has also taught me the importance of discipline, critical thinking, and perseverance in research and development. I shall always remain grateful to him for his mentorship and support throughout this phase of my academic journey.

Dipendu Dhali

Abstract

This thesis presents the design and development of a full-stack e-commerce web application focused on speed, security, and smooth performance even in poor network conditions.

The frontend uses Next.js with static site generation (SSG) and server-side rendering (SSR) to improve performance and search visibility. Tailwind CSS ensures the app looks clean and works well on all devices. React Icons add a visual touch.

The backend is built using Express.js, with a modular structure for easier development. Security is a top priority. The app uses JSON Web Tokens (JWT) and AES-256-GCM encryption to protect user data. Axios is used for efficient communication between frontend and backend, with support for error handling.

For payments, Razorpay is used to handle secure transactions within India. The app is deployed on Vercel to benefit from its fast and reliable global edge network.

Key features include protected routes, secure multi-step checkout, session handling, and a flexible API design. The project is a practical example of using modern tools and frameworks to build a fast, secure, and user-friendly e-commerce site.

Contents

List of Figures

Chapter 1
Introduction

1.1 Problem Statement

The contemporary e-commerce landscape is characterized by intense competition and rapidly evolving user expectations. Users demand platforms that are not only feature-rich but also exceptionally fast, secure, and accessible across a multitude of devices and network conditions [1]. Slow loading times, even delays of a single second, can significantly decrease conversion rates and negatively impact revenue [2]. Studies indicate that nearly half of online shoppers expect a webpage to load in two seconds or less, with conversion rates dropping dramatically as load times increase [2]. For instance, research found pages loading in 2.4 seconds achieved a 1.9% conversion rate, which fell below 1% at 4.2 seconds and to 0.6% beyond 5.7 seconds [2]. Conversely, performance improvements yield tangible benefits; Walmart observed a 2% conversion increase for every 1-second improvement in load time [2]. This underscores the critical need for performance optimization in e-commerce.

Furthermore, security remains a paramount concern. The constant threat of data breaches, payment fraud, and unauthorized access necessitates robust security measures [5]. E-commerce sites handle sensitive user data, including personal information and payment details, making them prime targets for cybercriminals [7]. Failure to adequately protect this data can lead to severe financial penalties, reputational damage, and loss of customer trust [5]. Research indicates that 60% of small businesses close within six months of a data breach, highlighting the existential threat posed by security lapses [8]. Compliance with standards like PCI DSS is not merely a regulatory hurdle but a fundamental requirement for building and maintaining customer confidence [5].

Finally, the user experience must be seamless and responsive, particularly on mobile devices, which dominate e-commerce traffic, especially in rapidly growing markets like India [10]. A poorly designed or unresponsive interface, particularly during the critical checkout process, can lead to high cart abandonment rates [12]. Challenges in building modern e-commerce platforms include navigating increasing competition, managing complex technological integrations like AI, ensuring cybersecurity and data privacy, maintaining supply chain resilience, and delivering consistent, personalized user experiences across all touchpoints [1].

This project directly addresses these challenges by proposing the development of an e-commerce platform that is:

- **Lightweight and Fast**: Utilizing Next.js for optimized rendering (SSR/SSG/ISR) and Vercel's Edge Network for efficient delivery, aiming to minimize load times even on poor networks [13].

- **Secure**: Implementing JWT-based authentication, AES encryption for sensitive data, secure coding practices (input validation, rate limiting), adherence to OWASP guidelines, and leveraging a PCI DSS compliant payment gateway (Razorpay) [5].
- **Responsive**: Employing Tailwind CSS for a mobile-first, utility-driven design approach to ensure a consistent and intuitive user experience across all devices [19].

The need for such a platform is particularly acute in markets like India, characterized by rapid e-commerce growth, high mobile penetration, diverse network quality, and increasing digital payment adoption [10]. Existing platforms may struggle to optimally serve users in Tier 2/3 cities or those with less stable internet connections. This project aims to demonstrate that a modern JavaScript stack, thoughtfully architected and deployed, can effectively meet these multifaceted demands. The platform's lightweight nature is crucial for accessibility in regions with infrastructural challenges or where users rely on lower-spec devices or limited data plans [10]. Optimizing for these conditions ensures broader market reach and caters to the significant user base emerging from non-metropolitan areas in India [10]. Therefore, "lightweight" directly supports capturing this growing market segment by ensuring the platform performs well even under suboptimal network conditions.

1.2 Project Significance and Motivation

The development of this full-stack e-commerce platform using Next.js and Express.js holds significant relevance in the current technological landscape. The JavaScript ecosystem, with Node.js enabling server-side execution, has become a dominant force in modern web development, allowing for unified language use across the entire stack [23]. Frameworks like Next.js (built on React) and Express.js represent contemporary choices within this ecosystem, offering powerful features for building performant and scalable applications [25]. Demonstrating their effective integration for a demanding application like e-commerce provides valuable insights into current best practices.

The project's context is further amplified by the phenomenal growth of the global e-commerce market. Global sales were projected to reach $6.3 trillion in 2024, continuing a strong upward trend [11]. The number of online shoppers globally surpassed 2.64 billion in 2023 and continues to grow [11]. Within this global expansion, the Indian e-commerce market stands out as one of the fastest-growing regions [10]. Valued at approximately $83–117.6 billion in 2024 [10], the Indian market is projected to experience a Compound Annual Growth Rate (CAGR) of 15% between 2024–2027 [10], or even higher according to some estimates (e.g., 39.37% CAGR from 2024–2029 reaching INR 53.42 trillion [33]). This rapid growth is fueled by increasing internet penetration (over 880 million users as of March 2023 [22]), widespread smartphone adoption (mobile accounting for 75–81% of e-commerce volume/purchases [10]), affordable data costs, and a young, tech-savvy demographic (one-third Gen Z shoppers [10]). The expansion into Tier 2 and Tier 3 cities, which now account for 60% of online orders [10], further underscores the need for platforms optimized for diverse user environments.

The motivation for this project stems directly from the need to address the critical challenges of performance, security, and responsiveness identified in the problem statement (Section 1.1). By leveraging the specific strengths of the chosen technology stack – Next.js for its rendering optimizations and SEO benefits [24], Express.js for its lightweight and flexible backend capabilities [27], Tailwind CSS for rapid and responsive UI development [19], JWT and AES for robust security [16], Axios for efficient communication [36], Razorpay for targeted payment

integration [37], and Vercel for optimized deployment [14] – this thesis aims to provide a concrete solution tailored to the demands of modern e-commerce, particularly within the dynamic Indian context.

1.3 Thesis Objectives

This thesis aims to achieve the following key objectives through the design, implementation, and evaluation of the e-commerce platform:

- **Develop a Full-Stack E-commerce Platform**: To architect and build a complete e-commerce application utilizing modern JavaScript frameworks, specifically Next.js for the frontend and Express.js for the backend, demonstrating the integration and capabilities of this stack for a real-world use case [24].
- **Implement Secure User Authentication**: To design and implement a robust and secure user authentication system employing JSON Web Tokens (JWT) for stateless session management and Advanced Encryption Standard (AES) for safeguarding sensitive user credentials and data, adhering to security best practices [16].
- **Create a Responsive, Mobile-First UI**: To develop a fully responsive user interface using Tailwind CSS, prioritizing a mobile-first design approach to ensure optimal viewing and interaction across a wide range of devices, from smartphones to desktops [19].

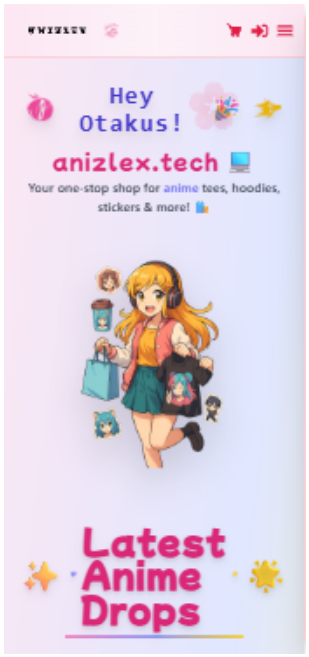


Figure 1.1: Responsive, Mobile-First UI:

- **Integrate a Real-Time Payment Gateway**: To integrate the Razorpay payment gateway, enabling secure and reliable online transaction processing tailored to the Indian payment ecosystem, including support for methods like UPI [10].
- **Optimize for Low-Bandwidth Environments**: To rigorously optimize application load times and user interface responsiveness, specifically targeting users operating under low-

bandwidth or unstable network conditions, thereby enhancing accessibility and user experience in diverse connectivity environments [40].

- **Utilize Axios for API Communication**: To employ the Axios library for managing HTTP requests between the Next.js frontend and the Express.js backend, leveraging features like interceptors for efficient global handling of requests, responses, and errors [36].
- **Enhance User Experience**: To improve the overall user journey through features such as dynamic routing for seamless navigation between pages (e.g., product details, categories), protected pages accessible only to authenticated users, and effective session management [46].
- **Apply Modular Coding Practices**: To implement the system using modular coding practices and clean architecture principles, promoting code reusability, testability, scalability, and long-term maintainability for both frontend and backend codebases [47].
- **Deploy on Vercel**: To successfully deploy the developed platform onto the Vercel infrastructure, leveraging its optimized environment for Next.js applications, global Edge Network, and automatic scaling capabilities to achieve production-level performance and global accessibility [14].
- **Showcase Modern Tool Integration**: To demonstrate the practical integration and effective use of various modern development tools and libraries, including React Icons for UI enhancement, the Razorpay SDK for payment processing, and environment-based configurations for managing settings across development and production stages.
- **Provide an Extensible Admin Structure**: To design and implement the backend API structure in a way that is admin-friendly and extensible, facilitating future development of administrative functionalities such as inventory management, order processing, and user administration, leveraging the already created backend API endpoints [48].

1.4 Thesis Structure

This thesis is organized into eight chapters, systematically detailing the development and analysis of the lightweight, secure, and responsive e-commerce platform.

- **Chapter 1: Introduction** presents the problem statement, highlighting the challenges in modern e-commerce concerning performance, security, and user experience. It establishes the project's significance within the growing global and Indian e-commerce markets and outlines the specific objectives guiding the research and development.
- **Chapter 2: Background and Literature Review** provides a comprehensive overview of the e-commerce landscape, delves into the chosen web development technologies (Next.js, Express.js, Tailwind CSS, etc.), discusses the critical importance of web performance and security (including JWT, AES, PCI DSS), introduces payment gateways (Razorpay), deployment platforms (Vercel), and Progressive Web Apps (PWAs).
- **Chapter 3: System Design and Architecture** details the architectural philosophy, justifies the technology stack selection, presents high-level system diagrams, and elaborates on the specific architectures for the frontend, backend, and database, including schema design and API principles.

- **Chapter 4: Implementation Details** describes the practical execution of the design, covering frontend development with Next.js and Tailwind, backend development with Express.js, database implementation, the JWT authentication flow, Razorpay payment gateway integration, client-server communication using Axios, and the structure of the admin panel APIs.
- **Chapter 5: Security Implementation and Analysis** focuses on the security aspects, analyzing the JWT authentication implementation against best practices, detailing AES encryption usage, outlining secure coding practices employed (input validation, rate limiting, headers), reviewing mitigation strategies against the OWASP Top 10, discussing payment security via Razorpay, and explaining the Role-Based Access Control (RBAC) mechanism.
- **Chapter 6: Deployment** details the deployment strategy using Vercel, describing the workflow, configuration, and how specific Vercel features were leveraged to enhance the application's performance, scalability, and reliability in a production environment.
- **Chapter 7: Conclusion and Future Work** summarizes the project's achievements, reiterates key findings and contributions regarding the effectiveness of the chosen stack and techniques, discusses limitations, and proposes potential avenues for future development and research.

Chapter 2
Background and Literature Review

This chapter provides the foundational context for the thesis, exploring the current state of e-commerce globally and specifically in India, examining the relevant web development technologies, and discussing the critical aspects of web performance, security, payment processing, deployment, and progressive web application principles that inform the project's design and objectives.

2.1 The E-commerce Landscape

The digital transformation has profoundly reshaped commerce, with online platforms becoming increasingly central to retail strategies worldwide. Understanding the dynamics of this landscape, both globally and within the target market of India, is crucial for contextualizing the development of a modern e-commerce platform.

2.1.1 Global E-commerce Growth and Trends

The global e-commerce market continues its robust expansion. Estimates projected global e-commerce sales to reach $6.3 trillion in 2024, representing a significant 9.4% annual increase from $5.8 trillion in 2023.[30] Forecasts suggest this growth will continue, potentially exceeding $8 trillion by 2027.[30] This growth is driven by an expanding base of online shoppers, estimated to reach 2.71 billion in 2024[11], and the increasing share of e-commerce within total retail sales, projected at 20.1% for 2024.[30]

Several key trends characterize the global market:

Mobile Commerce (M-commerce): Mobile devices are the dominant channel for online shopping. M-commerce sales were expected to reach $2.07 trillion in 2024, constituting a significant portion (around 60%) of total e-commerce transactions.[11] This highlights the imperative for mobile-first design strategies.

Cross-Border E-commerce: Consumers are increasingly purchasing from international markets. The global cross-border e-commerce market reached an estimated $791.5 billion in 2024, with projections indicating strong future growth.[30]

B2B E-commerce Growth: While this thesis focuses on B2C, the B2B e-commerce market is also experiencing massive growth, projected to reach $36 trillion by 2026, with the Asia-Pacific region expected to hold a dominant share.[30] Over 90% of B2B companies now utilize virtual sales models.[30]

Dominant Product Categories: Globally, categories like Fashion, Food, Consumer Electronics, and DIY/Hardware are projected to be major revenue drivers in the coming years, with Food e-commerce potentially becoming the largest segment by 2029.[31]

2.1.2 The Indian E-commerce Market

India represents one of the most dynamic and rapidly growing e-commerce markets globally, particularly within the Asia-Pacific region.[10] Despite infrastructural challenges, especially in rural areas, the sector is on a steep upward trajectory.[10]

Market Size and Growth: Estimates for India's e-commerce market value in 2024 vary but consistently point to substantial figures, ranging from INR 4.4 trillion ($53 billion using rough conversion) to $117.6 billion or even $182 billion (total volume).[10] Forecasts predict continued strong growth, with CAGR estimates ranging from 8.08% (2024-2028)[32] to 15% (2024-2027)[10], and even higher rates like 14.1% (2023-2027, ranking first globally)[51] or 39.37% (2024-2029).[33] The market is expected to reach values between $160.5 billion and $274 billion or INR 53.42 trillion by 2027-2029.[10] The number of online shoppers is also projected to surge, potentially exceeding 500 million by 2029.[22]

Shopper Demographics and Behavior: A significant driver is the young, tech-savvy population, with one-third of online shoppers being Gen Z.[10] Growth is increasingly fueled by consumers in Tier 2 cities and smaller towns, which now account for 60% of all online orders.[10] While average selling prices might be slightly lower in these areas compared to metros, their sheer volume is transforming the market.[22] Indian shoppers prioritize factors like higher quality (40%) and product availability (33%) when shopping online.[10] Groceries are the most frequently purchased category.[10] There's also a growing trend towards social commerce, projected to be worth $37 billion by 2025[10], and cross-border purchases, with the US, Australia, and China being top sources.[10] Rising affluence is another key factor, with 60-70 million households expected to enter upper-middle/upper-income brackets by 2024, projected to drive 85% of e-retail GMV by 2028.[10]

Device Usage: Mobile is paramount. Approximately 81% of Indian e-commerce shoppers use smartphones[10], and mobile devices account for 75% of e-commerce transaction volume.[10] This necessitates a mobile-first design approach.

Payment Methods: The Unified Payment Interface (UPI) dominates the payment landscape, accounting for 55% of e-commerce transaction volume.[10] Credit cards (25%) follow, with digital wallets, cash on delivery, net banking, BNPL, and debit cards making up smaller shares.[10] Integrating payment gateways that efficiently support UPI, like Razorpay, is therefore critical.[37]

Market Players and Investment: The market is led by major players like Amazon and Flipkart, followed by others like Myntra and IndiaMART.[10] Significant investment continues to flow into the sector, with established players like Flipkart raising substantial funds ($1 billion target, $600M from Walmart) to expand reach, particularly in smaller towns, and newer startups like BuyEazzy also securing funding for expansion.[22]

The unique characteristics of the Indian e-commerce market – its rapid growth fueled by mobile adoption, the rise of Tier 2/3 cities, the dominance of UPI payments, and the increasing number of digitally native shoppers – create a specific context. This context strongly motivates the development of platforms optimized for these conditions. A lightweight, responsive platform designed for mobile-first access, capable of performing well under variable network conditions

often found outside major metros, and seamlessly integrating with popular payment methods like UPI via gateways such as Razorpay, is precisely aligned with the key growth drivers and user needs within this dynamic market.[10] The objectives of this thesis project, particularly those focusing on mobile-first UI (Objective #3), low-bandwidth optimization (Objective #5), and Razorpay integration (Objective #4), directly address these market realities, positioning the developed platform to effectively serve this burgeoning user base.

2.2 Modern Web Development Technologies

The choice of technology stack significantly influences an application's performance, scalability, security, and maintainability. This project leverages a modern JavaScript-centric stack, reflecting current trends in full-stack development.

2.2.1 JavaScript Ecosystem

JavaScript has evolved from a client-side scripting language to a versatile language capable of powering both frontend and backend development. The advent of Node.js enabled server-side JavaScript execution, fostering a rich ecosystem of frameworks, libraries, and tools.[23] This allows development teams to potentially use a single language across the entire application stack, which can streamline development processes and facilitate code sharing.[24]

2.2.2 React and Next.js

React is a widely adopted JavaScript library for building user interfaces, known for its component-based architecture and virtual DOM, which facilitates efficient UI updates.[24] However, standard React applications primarily render on the client-side (CSR), which can pose challenges for initial load performance and Search Engine Optimization (SEO) as content is often not immediately available to search engine crawlers.[13]

Next.js is a popular framework built on top of React, designed to overcome these limitations and provide a more robust structure for building production-ready web applications.[24] Key benefits offered by Next.js include:

Flexible Rendering Strategies: Next.js provides built-in support for Server-Side Rendering (SSR), Static Site Generation (SSG), and Incremental Static Regeneration (ISR) on a per-page basis.[13] SSR generates HTML on the server for each request, ensuring up-to-date content and excellent SEO.[13] SSG pre-renders HTML at build time, offering maximum performance for static content.[13] ISR combines static generation with periodic or on-demand regeneration, balancing performance and freshness, particularly useful for large sites.[13] This flexibility is crucial for e-commerce sites needing both fast static pages and dynamic, SEO-friendly product pages.[28]

File-System Based Routing: Simplifies route creation by mapping files and folders within the pages (or app) directory to URL paths, including support for dynamic routes.[24]

API Routes: Allows developers to build backend API endpoints directly within the Next.js project structure (e.g., under pages/api/ or app/api/), simplifying full-stack development for certain use cases.[26]

Built-in Optimizations: Includes automatic code splitting (loading only necessary JavaScript per page)[15], image optimization (next/image)[14], and font optimization (next/font).[14]

Developer Experience: Features like Fast Refresh (Hot Module Replacement) enhance productivity.[25]

While powerful, Next.js can present a steeper learning curve than basic React, requiring an understanding of its specific rendering methods and conventions.[25] Configuration for advanced features can also be complex.[25] There might also be a perceived ”lock-in” with Vercel, its parent company, as some features are best optimized for that platform.[26]

2.2.3 Node.js and Express.js

Node.js provides an asynchronous, event-driven JavaScript runtime environment built on Chrome's V8 engine.[23] Its non-blocking I/O model makes it well-suited for building scalable, high-performance network applications, including web servers and APIs.[23]

Express.js is a minimal and flexible Node.js web application framework, often considered the de facto standard for building backend services and APIs with Node.js.[23] Its core strengths lie in:

Middleware Pipeline: Express operates on a middleware system, where functions process requests sequentially.[23] Middleware can execute code, modify request/response objects, end the cycle, or pass control to the next middleware using the next() function.[23] This allows for modular implementation of concerns like logging, authentication, authorization, input validation, and error handling.[27] Express supports various middleware types: application-level, router-level, error-handling, built-in (like express.json() for body parsing, express.static() for serving static files), and third-party.[27]

Routing: Provides a simple yet powerful system for defining how the application responds to client requests based on URL paths and HTTP methods (GET, POST, PUT, DELETE, etc.).[23] It supports route parameters, string patterns, and regular expressions for flexible path matching.[34] The express.Router class facilitates modular routing, allowing related routes to be grouped and managed separately.[47]

Performance: Leverages Node.js's performance characteristics for building fast APIs.[23]

Flexibility and Unopinionated Nature: Express provides a minimal core, allowing developers to structure their application and choose supporting libraries as needed.[23]

Large Ecosystem and Community: Benefits from the vast Node.js ecosystem (npm) and has a large, active community providing extensive middleware, documentation, and support.[23]

However, Express's minimalism means developers are responsible for defining the application structure and integrating necessary components for tasks beyond basic routing and middleware.[29] Managing complex middleware chains can also become challenging in large applications.[34] Compared to more opinionated frameworks like Python's Django or PHP's Laravel, Express offers less built-in functionality (e.g., ORM, admin panels) but greater flexibility.[29]

2.2.4 Tailwind CSS

Tailwind CSS is a utility-first CSS framework that provides low-level utility classes for styling HTML elements directly within the markup.[19] This contrasts with traditional CSS methodologies (writing custom classes in separate files) and component-based frameworks like Bootstrap (which offer pre-designed components).[27]

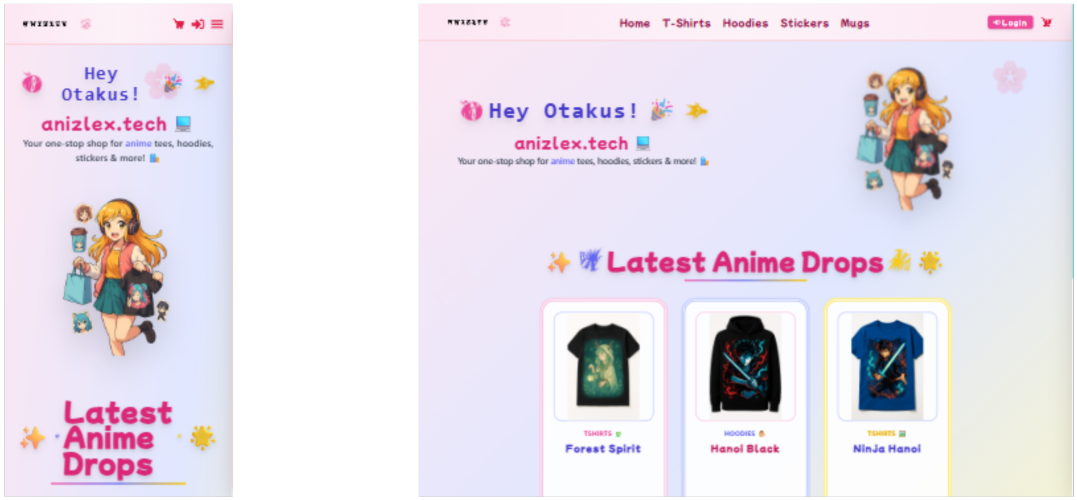
Key advantages of Tailwind's approach include:

Rapid Development: Applying styles directly in HTML speeds up the development process by reducing the need to switch contexts or invent class names.[19]

High Customizability: Developers have granular control and can build completely custom designs without fighting framework opinions or writing extensive CSS overrides.[19] The framework is highly configurable via tailwind.config.js.[19]

Design Consistency: Using a predefined set of utility classes based on a configurable design system promotes visual consistency across the application.[20]

Responsiveness: Built-in responsive variants (e.g., sm:, md:, lg:) make implementing responsive layouts intuitive and straightforward, often following a mobile-first pattern.[19]



Mobile View PC View

Figure 2.1: Responsive UI using tailwind

Performance (Reduced Bloat): Tailwind utilizes tools like PurgeCSS or its integrated Just-In-Time (JIT) engine to scan project files and generate only the CSS corresponding to the utility classes actually used. This results in significantly smaller final CSS bundles compared to including a full component library or accumulating unused custom CSS.[19]

Tailwind's utility-first philosophy aligns well with the project's goal of creating a custom, responsive UI efficiently, without the constraints imposed by pre-built component libraries. While it might initially seem verbose in the HTML, proponents argue this leads to more maintainable and scalable styling in the long run. Component libraries built with Tailwind (like Tailwind UI or DaisyUI) can offer pre-built elements while still leveraging Tailwind's utility classes for customization.[56]

The combination of Next.js and Express.js presents a modern, potentially high-performing JavaScript-centric architecture. Next.js addresses frontend performance and SEO challenges through its advanced rendering capabilities, while Express.js offers a flexible foundation for building the backend API. However, this stack requires deliberate architectural planning. Express's unopinionated nature necessitates careful structuring of the backend codebase (e.g.,

modular routing, service layers)[29], and developers must be proficient with Next.js's specific rendering paradigms and routing conventions.[25] Tailwind CSS complements this stack by enabling the rapid development of a custom, consistent, and responsive user interface without the limitations or potential bloat of traditional component libraries[19], directly supporting the project's responsiveness objective (Objective #3). This careful selection and integration of technologies form the basis for achieving the project's goals of building a lightweight, secure, and responsive platform.

2.3 Web Performance and User Experience

Web performance is not merely a technical metric; it is a fundamental aspect of user experience (UX) and a critical driver of business success, particularly in the competitive e-commerce domain. Slow performance directly translates to user frustration, higher bounce rates, and lost revenue.

2.3.1 Defining Web Performance

Web performance is typically measured using a set of metrics known as Core Web Vitals, along with other indicators. Key metrics include:

Largest Contentful Paint (LCP): Measures loading performance – the time it takes for the largest image or text block visible within the viewport to render. A good LCP is generally considered 2.5 seconds or less.[59]

First Contentful Paint (FCP): Measures the time from when the page starts loading to when any part of the page's content is rendered on the screen. Optimizing FCP has shown tangible conversion benefits.[57]

Interaction to Next Paint (INP): Measures responsiveness – the time it takes for the page to respond to user interactions (like clicks or key presses). This replaced First Input Delay (FID) as a Core Web Vital in March 2024.[59] Good INP is typically below 200 milliseconds.[59]

Cumulative Layout Shift (CLS): Measures visual stability – the amount of unexpected layout shift of visible page content. A good CLS score is 0.1 or less.[59]

Time to First Byte (TTFB): Measures server responsiveness – the time between the browser requesting a page and when it receives the first byte of information from the server.[52]

2.3.2 Impact on Conversion Rates

The link between website performance and e-commerce conversion rates is direct and well-documented.[2] Users have high expectations; studies show 47% expect pages to load in 2 seconds or less, and over 60% of smartphone users want load times under 3 seconds.[2] Delays have a significant negative impact:

A 1-second delay can lead to a 7% reduction in conversions.[3]

Pages loading in 2.4 seconds had a 1.9% conversion rate, dropping below 1% at 4.2 seconds.[2]

Websites loading in 1 second have conversion rates 2.5x higher than those taking 5 seconds[58], and 3x higher than 5-second loads / 5x higher than 10-second loads according to another source.[57]

Bounce rates increase significantly with load time; pages loading in 3 seconds have a 32% higher bounce rate than 1-second loads.[57] The probability of bounce increases by 90% for a 5-second load time compared to a 1-second load time.[60]

Conversely, improvements drive conversions: Walmart found a 2% conversion increase per 1-second improvement.[2] COOK saw a 7% conversion increase by reducing load time by 0.85 seconds.[2] Mobify found a 1.11% conversion increase per 100ms improvement.[2] Deloitte found a 0.1-second improvement boosted retail conversions by 8.4%.[57]

These statistics clearly demonstrate that optimizing performance is not just a technical exercise but a crucial business strategy in e-commerce. Even seemingly small improvements in load time can translate into substantial increases in user engagement, conversions, and ultimately, revenue.[2] This direct link between speed and business success validates the project's focus on performance optimization (Objective #5) as a core requirement for building a viable e-commerce platform.

2.3.3 Optimization Techniques Overview

Achieving optimal web performance requires a multi-faceted approach involving various techniques applied across the application stack. Some key areas include:

Efficient Rendering Strategies: Choosing the right rendering method (SSR, SSG, ISR in Next.js) for different types of content to balance initial load speed, dynamic data needs, and SEO.[13]

Caching: Implementing caching at multiple levels (browser, CDN, server-side data cache, application cache) to store frequently accessed data and assets, reducing load times and server strain.[4]

Code Splitting & Minification: Breaking down JavaScript and CSS into smaller chunks loaded only when needed (code splitting) and removing unnecessary characters from code (minification) to reduce file sizes.[4]

Image Optimization: Compressing images, using modern formats (like WebP), and serving appropriately sized images for different devices (responsive images).[4]

Reducing HTTP Requests: Minimizing the number of requests the browser needs to make by combining files (CSS, JS) where appropriate.[4]

Server Optimization: Reducing server response time through efficient backend logic, database query optimization, and potentially using Content Delivery Networks (CDNs).[4]

Third-Party Script Management: Loading external scripts (analytics, ads, chat widgets) efficiently using techniques like async, defer, or lazy loading to avoid blocking page rendering.[42]

These techniques, explored in detail in Chapter 6, are essential for meeting user expectations and achieving the performance goals of the e-commerce platform.

2.4 Web Security Imperatives

Security is non-negotiable in e-commerce. The financial and reputational consequences of security failures can be catastrophic.[5] Building a secure platform requires a proactive approach,

addressing common vulnerabilities and implementing robust protective measures throughout the development lifecycle.

2.4.1 Common E-commerce Threats

E-commerce platforms are attractive targets due to the sensitive data they handle. Common threats include:

Data Breaches: Unauthorized access to sensitive data like Personally Identifiable Information (PII), login credentials, and payment details. Weak or stolen credentials are a frequent cause.[6]

Payment Fraud: Unauthorized transactions using stolen card details or other fraudulent means. Global payment fraud losses were estimated at $48 billion in 2023.[6]

Injection Attacks: Exploiting vulnerabilities by injecting malicious code (SQL, NoSQL, Cross-Site Scripting (XSS), Command Injection) through user inputs to manipulate the application or database.[61]

Broken Authentication/Authorization: Flaws allowing attackers to bypass login mechanisms, hijack sessions, or gain unauthorized access to functions or data.[62]

Denial-of-Service (DoS/DDoS) Attacks: Overwhelming the server with traffic to make the application unavailable to legitimate users.[18]

Security Misconfiguration: Vulnerabilities arising from insecure default settings, unpatched software, or improperly configured security controls.[63]

Using Components with Known Vulnerabilities: Relying on outdated or insecure third-party libraries or frameworks.[62]

Server-Side Request Forgery (SSRF): Tricking the server into making requests to unintended internal or external resources.[63]

The OWASP Top 10 list provides a widely recognized framework for understanding and prioritizing the most critical web application security risks.[61] Addressing these risks is fundamental to building a secure e-commerce platform.

2.4.2 Importance of Data Security (PCI DSS)

The Payment Card Industry Data Security Standard (PCI DSS) is a set of security standards mandated by major credit card brands (Visa, Mastercard, Amex, Discover, JCB) for any organization that accepts, processes, stores, or transmits cardholder data. Compliance involves implementing specific technical and operational requirements, including building and maintaining secure networks, protecting cardholder data (e.g., through encryption), maintaining a vulnerability management program, implementing strong access control measures, regularly monitoring and testing networks, and maintaining an information security policy.[5]

For e-commerce businesses, PCI DSS compliance is crucial[5]:

- Reduces Risk: Helps protect sensitive payment information from breaches and fraud.[5]
- Builds Trust: Demonstrates a commitment to security, boosting customer confidence and loyalty.[5] Nearly 15% of consumers abandon purchases due to security concerns.[9]

- Avoids Penalties: Non-compliance can result in significant fines (ranging from $5,000 to $100,000 monthly), suspension of payment processing capabilities, and legal repercussions.[5]
- Meets Legal Requirements: Compliance often intersects with data privacy laws like CCPA.[5]

Using a PCI DSS compliant payment gateway like Razorpay significantly simplifies compliance for merchants by outsourcing the direct handling and storage of sensitive card data.[5] However, merchants still have responsibilities regarding the security of their own systems and integration points.

2.4.3 Authentication and Authorization Concepts

Distinguishing between authentication and authorization is fundamental to security design[46]:

Authentication: The process of verifying a user's identity – proving they are who they claim to be (e.g., using username/password, MFA).[46]

Authorization: The process of determining what actions an authenticated user is allowed to perform or what resources they can access.[46]

Token-Based Authentication (JWT): JSON Web Tokens (JWTs) are a standard (RFC 7519) for creating access tokens that assert claims between two parties.[16] They are commonly used for stateless authentication in APIs and web applications.[39] A JWT typically consists of three parts: header (metadata, algorithm), payload (claims about the user, expiration time), and signature (to verify integrity).[16] The server issues a signed JWT upon successful login, and the client sends this token with subsequent requests to prove identity.[16] Secure implementation requires careful handling of signing algorithms, key management, token validation, expiration, and storage.[16]

Role-Based Access Control (RBAC): RBAC is an authorization strategy where access permissions are assigned based on user roles (e.g., 'admin', 'user', 'moderator') rather than individual users.[50] This simplifies permission management, especially in applications with many users and varying levels of access.[50] Implementation typically involves defining roles, assigning roles to users, and checking a user's role before granting access to specific resources or functionalities, often via middleware.[50]

2.4.4 Encryption Standards (AES)

Encryption is vital for protecting sensitive data both when it is stored (at rest) and when it is transmitted (in transit).[35] The Advanced Encryption Standard (AES) is the globally recognized standard for symmetric key encryption, adopted by the U.S. government and widely used across industries.[17]

Symmetric Key & Block Cipher: AES uses the same secret key for both encryption and decryption.[17] It operates as a block cipher, encrypting data in fixed-size blocks of 128 bits.[17]

Key Lengths: AES supports three key lengths: 128-bit, 192-bit, and 256-bit. Longer keys provide exponentially higher security against brute-force attacks but require more computational resources.[17] AES-256 is considered the most secure and is often required for highly sensitive data or stringent regulatory compliance.[35]

Operation: AES uses a substitution-permutation network, applying multiple rounds of transformations (SubBytes, ShiftRows, MixColumns, AddRoundKey) to the data block, with the number of rounds depending on the key size (10 for AES-128, 12 for AES-192, 14 for AES-256).[17]

Modes of Operation: Block ciphers like AES require a mode of operation to handle data larger than a single block. Common modes include CBC (Cipher Block Chaining) and CTR (Counter Mode). However, for modern applications requiring confidentiality and integrity, authenticated encryption modes like GCM (Galois/Counter Mode) or CCM (Counter with CBC-MAC) are strongly recommended.[71] GCM, in particular, is widely used (e.g., in TLS 1.2+) and offers high performance, especially with hardware acceleration.[71] NIST specifically recommends GCM for its efficiency and security properties.[72]

Applications: AES is ubiquitous in securing digital information, including HTTPS/TLS for secure web communication, VPNs, Wi-Fi (WPA2/WPA3), file and disk encryption, database encryption, and secure password storage (hashing is preferred, but AES might be used in specific contexts).[17] Enforcing HTTPS (which uses TLS, often with AES) for all application traffic is a fundamental security best practice.[16]

2.5 Payment Gateway Integration

Secure and reliable payment processing is the lifeblood of any e-commerce platform. Payment gateways act as essential intermediaries, facilitating the complex flow of information between the customer, the merchant, the acquiring bank, and the issuing bank.

2.5.1 Role of Payment Gateways

A payment gateway is a service that authorizes and processes online payments for e-businesses.[37] It acts as a secure bridge, encrypting sensitive information like credit card details and transmitting it securely between the merchant's website/app and the payment processor or bank.[37] Key functions include capturing payment details, encrypting data, routing transaction information to the appropriate networks, receiving authorization responses, and communicating the transaction status back to the merchant and customer.[74] They are crucial for enabling merchants to accept various payment methods securely and efficiently.[37]

2.5.2 Razorpay Overview

Razorpay is a leading payment solutions provider in India, offering a suite of products including a popular payment gateway.[37] It is particularly well-suited for the Indian market due to its extensive support for local payment methods.[37]

Key features and benefits relevant to this project include:

Wide Range of Payment Methods: Supports domestic and international credit/debit cards, EMIs (card-based and cardless), PayLater options, Netbanking from numerous banks, UPI (critical in India), and multiple mobile wallets.[37]

Developer-Friendly Integration: Provides robust, clean APIs and SDKs for various platforms and languages (including Node.js and React Native), simplifying the integration process.[37] Documentation and sample applications are available.

Security: Razorpay is PCI DSS Level 1 compliant, the highest level of certification, ensuring secure handling of card data.[37] They also employ frequent third-party audits and internal security teams.[37] Integration involves security measures like signature verification to authenticate payment responses.[74]

Checkout Experience: Offers customizable checkout options (Standard Checkout, Custom Checkout) designed for seamless user experience and potentially higher conversion rates.[37] Features like saved cards and native OTP handling further streamline the process.[37]

Dashboard and Reporting: Provides a powerful dashboard for monitoring payments, settlements, refunds, and accessing detailed statistics for business insights.[37]

Webhooks: Supports webhooks for real-time notifications of payment events (e.g., payment captured, failed, refunded), enabling automated backend updates.[75]

Integrating a reputable and feature-rich gateway like Razorpay addresses Objective #4 and contributes significantly to the platform's security and user experience.

2.6 Deployment Platforms (Vercel)

The deployment platform plays a critical role in an application's performance, scalability, and availability. Modern platforms often leverage serverless computing and edge networks to optimize delivery.

2.6.1 Introduction to Serverless and Edge Computing

Serverless computing allows developers to run backend code without managing the underlying server infrastructure. Functions are executed in response to events, and the platform handles scaling automatically.[81] Edge computing involves running compute tasks closer to the end-user at the "edge" of the network, typically within a Content Delivery Network (CDN). This reduces latency by minimizing the physical distance data needs to travel.[38]

2.6.2 Vercel Platform Overview

Vercel is a cloud platform specifically designed for frontend developers, offering seamless deployment and hosting optimized for frameworks like Next.js (Vercel is the creator of Next.js).[14] Its key features relevant to this project include:

Optimized Next.js Support: Provides zero-configuration deployment for Next.js applications, automatically handling build processes and infrastructure setup. It fully supports Next.js features like SSR, SSG, ISR, API Routes, Image Optimization, and Font Optimization.[14]

Git Integration: Integrates directly with Git providers (GitHub, GitLab, Bitbucket) for automatic deployments on push and generation of preview URLs for every pull request, streamlining the development and review workflow.[14]

Global Edge Network: Vercel operates a global Edge Network consisting of Points of Presence (PoPs) and compute regions.[82] This network acts as a CDN, caching static assets and optimized images/fonts close to users worldwide. It also supports running Edge Functions (for middleware) and Vercel Functions (Serverless Functions for SSR/API routes) in regions close to users or data sources, minimizing latency.[14]

Automatic Scaling: Serverless Functions and Edge Functions scale automatically based on traffic demands, scaling down to zero when not in use, which can be cost-effective.[14]

Enhanced ISR/SSR: Vercel enhances Next.js ISR by providing global distribution and persistent storage for generated pages, enabling faster global updates (around 300ms). SSR is handled efficiently via scalable Serverless Functions.[14]

Security Features: Includes automatic HTTPS/SSL provisioning, DDoS protection, and integration points for authentication.[38]

Developer Experience: Known for its ease of use and focus on developer productivity.[38]

While alternatives like AWS offer greater customization and potentially lower costs at extreme scale, they typically require significantly more configuration and DevOps expertise.[38] Vercel's tight integration with Next.js and its focus on performance and developer experience make it a strong choice for deploying this project (Objective #9).

2.7 Progressive Web Apps (PWAs)

Progressive Web Apps (PWAs) represent a set of capabilities that allow web applications to provide a more native app-like experience, including offline functionality, push notifications, and installability.

2.7.1 PWA Concepts

PWAs are web applications built using modern web technologies that aim to deliver a reliable, fast, and engaging user experience.[43] Key characteristics include:

- **Responsive:** Adapt to any form factor (desktop, mobile, tablet).[43]
- **Connectivity Independent:** Function offline or on low-quality networks using service workers.[43]
- **App-like:** Offer navigation and interactions similar to native apps, potentially running full-screen.[43]
- **Installable:** Users can add the PWA to their home screen or app drawer.[44]
- **Fresh:** Automatically update via the service worker update process.[43]
- **Secure:** Served over HTTPS.[44]
- **Discoverable:** Identifiable as "applications" thanks to web manifest files.[84]

2.7.2 Role of Service Workers

Service workers are JavaScript scripts that run in the background, separate from the web page, acting as a proxy between the browser and the network.[43] They are the core technology enabling key PWA features:

Offline Caching: Service workers can intercept network requests and serve responses from a cache, allowing the PWA to work offline or load faster on repeat visits.[41] Various caching strategies (e.g., cache-first, network-first, stale-while-revalidate) can be implemented.[43]

Push Notifications: Enable the application to receive push messages from a server, even when the PWA is not active.[43]

Background Sync: Allow tasks like data synchronization to occur in the background, even if the user navigates away or closes the app.[43]

2.7.3 Relevance to E-commerce

PWA capabilities are highly relevant for e-commerce, particularly in markets with variable network quality like India:

- **Improved Performance:** Caching strategies powered by service workers can significantly reduce load times, especially for returning visitors or users on slow networks.[41]
- **Offline Access:** Allows users to browse previously visited products or categories even without an internet connection, enhancing reliability and engagement.[43]
- **Engagement:** Push notifications can be used for order updates, promotions, or re-engagement campaigns.[43]
- **Installability:** Adding the app to the home screen provides easier access and a more integrated experience.[44]

Implementing PWA features, especially service worker caching, directly addresses Objective #5 (Optimize for low-bandwidth environments) by making the application more resilient and performant under challenging network conditions.[41] Libraries like next-pwa can simplify the integration of service workers and manifest files into a Next.js application.[83]

Table 2.1: **Key E-commerce Market Statistics (India & Global)**

Metric	**Global Value/Rate (2024 Est.)**	**India Value/Rate (2024 Est.)**	**Source(s)**
Market Size (Retail)	$6.3 Trillion	$83 – $118 Billion	[10]
Annual Growth Rate	9.4%	8% – 15% (CAGR varies)	[10]
Mobile E-commerce Volume	∼60% (Sales)	∼75% (Volume)	[10]
UPI Payment Share	N/A	55%	[10]
Orders from Tier 2/3 Cities	N/A	60%	[10]
Online Shoppers	2.71 Billion	∼500 Million (by 2029)	[11]

Note: Figures are estimates based on cited sources and may vary slightly depending on the specific report and methodology. Indian market size converted roughly where necessary.

Chapter 3
System Design and Architecture

This chapter outlines the architectural blueprint for the e-commerce platform. It details the overall design philosophy, provides a rationale for the chosen technology stack, presents the system architecture visually, and elaborates on the specific designs for the frontend, backend, database, and API layers. The architecture is guided by the project's core objectives: creating a lightweight, secure, responsive, and maintainable platform using modern JavaScript technologies.

3.1 Architectural Philosophy

The core architectural decision was to adopt a full-stack JavaScript approach, utilizing Next.js for the frontend and Express.js for the backend. This contrasts with alternatives such as using Next.js's built-in API routes for the entire backend, employing a traditional MERN (MongoDB, Express.js, React, Node.js) stack where React is used more directly without Next.js's framework features, or opting for different backend languages/frameworks like Python/Django or PHP/Laravel.[24]

The full-stack JavaScript approach was chosen for several potential benefits:

- **Unified Language:** Using JavaScript across the stack can potentially improve developer productivity and allow for easier code sharing or context switching.[23]
- **Ecosystem Alignment:** Leverages the vast and active Node.js/JavaScript ecosystem (npm) for both frontend and backend development.[23]
- **Modern Framework Capabilities:** Utilizes Next.js's advanced rendering features (SSR/SSG/ISR) for frontend performance and SEO, which are often more integrated than adding SSR capabilities to a standard React/Express setup.[24] Express.js provides a proven, flexible foundation for building the dedicated backend API.[27]

However, this approach also requires careful consideration. Express.js, being minimal, necessitates deliberate structuring of the backend application (e.g., defining layers for controllers, services, data access).[29] Next.js introduces its own conventions for routing and rendering that must be understood and managed.[25] The separation between the Next.js frontend server (handling page rendering and potentially some API routes) and the dedicated Express.js backend server (handling core business logic and data operations) needs clear definition and efficient communication protocols (handled via Axios in this project).

The design philosophy emphasizes the project's primary goals:

- **Lightweight:** Choosing Express.js for the backend contributes to this goal due to its minimal core.[27] Frontend performance optimizations (Chapter 6) further support this.
- **Responsive:** Addressed primarily through the use of Tailwind CSS and mobile-first design principles in the frontend architecture.[19]
- **Secure:** Security considerations permeate the design, from authentication (JWT) and encryption (AES) to secure coding practices and infrastructure choices (HTTPS, Vercel security features).[16]
- **Modularity and Clean Architecture:** Both frontend and backend are structured modularly to promote maintainability, testability, and scalability, aligning with Objective #8.[47]

3.2 Technology Stack Rationale (Deep Dive)

The selection of each component in the technology stack was driven by its specific capabilities and alignment with the project's objectives.

3.2.1 Frontend (Next.js)

Chosen over standard React primarily for its built-in solutions for performance and SEO challenges inherent in client-side rendering.[24] Its support for SSR, SSG, and ISR allows for optimizing different page types (e.g., fast static landing pages via SSG, dynamic SEO-friendly product pages via SSR or ISR).[13] Features like file-based routing, automatic code splitting, and integrated image/font optimization significantly enhance developer experience and application performance.[14] Its suitability for e-commerce applications is well-recognized.[28]

3.2.2 Backend (Express.js)

Selected for its minimalism, performance, and flexibility within the Node.js ecosystem.[23] Its robust middleware system is ideal for implementing cross-cutting concerns like authentication, logging, validation, and error handling in a modular way.[23] While Next.js API routes could handle simple backend tasks, a dedicated Express server provides greater control, separation of concerns, and scalability for the core business logic and database interactions of a full-featured e-commerce platform.[28] Its widespread adoption ensures ample community support and resources.[23]

3.2.3 Styling (Tailwind CSS)

Chosen for its utility-first approach, which enables rapid development of custom user interfaces without being constrained by pre-built components.[19] It promotes design consistency through a configurable utility system and significantly reduces CSS bundle size via purging unused styles.[19] Its inherent support for responsive design through breakpoint prefixes aligns perfectly with the mobile-first requirement (Objective #3).[19]

3.2.4 Database (MongoDB - Assumed)

Assuming MongoDB based on MERN comparisons and suitability for flexible product catalogs. A NoSQL database like MongoDB offers flexibility in handling potentially diverse and

evolving product structures (variants, attributes).[85] While SQL databases provide strong consistency guarantees ideal for transactional data (users, orders), NoSQL can offer performance and scalability advantages for read-heavy operations like displaying product catalogs.[85] A hybrid approach (SQL for users/orders, NoSQL for products) is often considered optimal[85], but for simplicity within a unified JavaScript stack, MongoDB provides a viable option, requiring careful application-level schema design and validation.[86] (Self-correction: The abstract/objectives don't specify the DB. While MongoDB is plausible given the stack context, this section should ideally state the actual choice and justify it, or acknowledge the assumption). For this thesis, we will proceed assuming MongoDB was chosen for its flexibility with product data and alignment with the JavaScript ecosystem.

3.2.5 API Communication (Axios)

Selected as a popular, promise-based HTTP client for both browser and Node.js.[36] Its key advantage lies in interceptors, allowing global configuration for requests (e.g., automatically attaching JWT Authorization headers) and responses (e.g., centralized error handling for different HTTP status codes, handling token expiry/refresh).[36] This promotes cleaner code in components making API calls (Objective #6).

3.2.6 Authentication (JWT)

Chosen for its suitability for stateless authentication, common in API-driven architectures.[16] JWTs allow the server to verify user identity without maintaining session state, improving scalability.[39] Secure implementation involves using strong signing algorithms, validating signatures and claims server-side, setting appropriate expirations, and storing tokens securely on the client (e.g., HttpOnly cookies)[16] (Objective #2).

3.2.7 Encryption (AES)

AES, specifically AES-256, is the industry standard for symmetric encryption, chosen for its proven security and performance.[17] Using an authenticated encryption mode like GCM (AES-256-GCM) is recommended to ensure both confidentiality and data integrity.[71] This will be applied to encrypt specific sensitive data fields where necessary (Objective #2).

3.2.8 Payment (Razorpay)

Selected due to its strong presence in the Indian market, comprehensive support for local payment methods (especially UPI), developer-friendly APIs and SDKs, and PCI DSS compliance[10] (Objective #4).

3.2.9 Deployment (Vercel)

Chosen for its seamless integration with Next.js, providing zero-configuration deployment, automatic scaling, a global Edge Network for performance, and built-in support for Next.js features like ISR, SSR, and Image Optimization[14] (Objective #9).

3.3 System Architecture Diagram

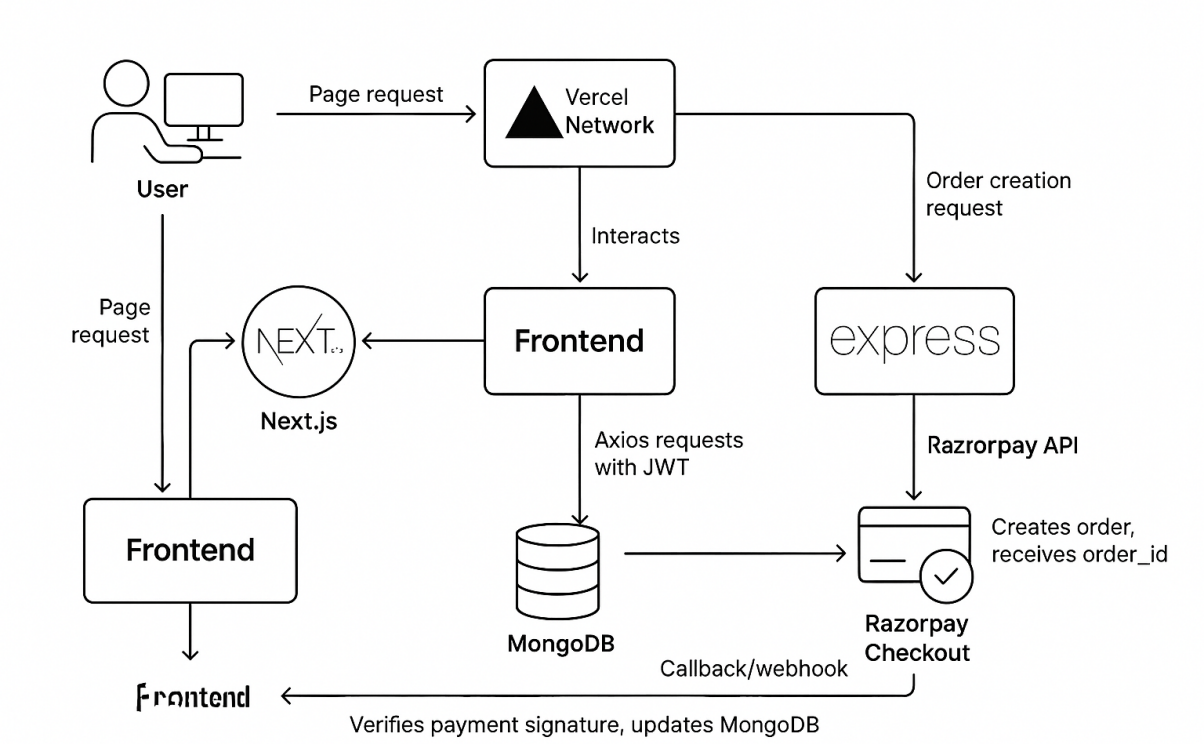


Figure 3.1: System Architecture Diagram

The system architecture comprises the following key components:

- **Client (Browser/Device):** User interacts with the application via a web browser on desktop or mobile.
- **Frontend (Next.js on Vercel):** Serves the user interface, rendered using a mix of SSG, SSR, and ISR. Handles client-side logic, state management, and routing. Makes API calls to the backend using Axios. Interacts with Razorpay Checkout for payment processing. Hosted on Vercel's Edge Network.
- **Backend (Express.js API):** Handles business logic, interacts with the database, manages user authentication/authorization (JWT verification), processes API requests from the frontend, and integrates with the Razorpay server-side SDK for order creation and payment verification. Likely hosted separately (e.g., on Vercel Functions, AWS Lambda, or a traditional server).
- **Database (MongoDB - Assumed):** Stores application data, including user information, product catalogs, orders, etc.
- **Razorpay:** External payment gateway service handling the actual payment processing and providing callbacks/webhooks to the backend.
- **Vercel Platform:** Provides hosting, CDN, serverless functions (for Next.js SSR/API routes and potentially the Express API), edge caching, and deployment infrastructure.

Interaction Flow (Simplified):

User requests a page → Vercel Edge Network serves cached static content or routes to Next.js (running on Vercel Functions).

Next.js renders the page (SSR/SSG/ISR), potentially fetching data from the Express API via Axios.

User interacts (e.g., adds to cart, logs in) → Frontend sends requests to Express API via Axios (with JWT if authenticated).

Express API processes request, interacts with MongoDB, verifies JWT via middleware.

Checkout → Frontend requests order creation from Express API → Express API interacts with Razorpay API → Razorpay returns order_id → Frontend initiates Razorpay Checkout with order_id → User pays on Razorpay → Razorpay sends callback/webhook to Express API → Express API verifies payment signature and updates MongoDB.

3.4 Frontend Architecture

The frontend is built using Next.js, leveraging its features for performance, SEO, and developer experience.

Project Structure: Utilizes the Next.js file-system based routing (pages or app directory) to define application routes. Components are organized modularly (e.g., by feature or type like components/ui, components/product, components/checkout).

Rendering Strategy: Employs a hybrid approach:

- **SSG:** For static content like landing pages, About Us, Contact Us, and potentially category listing pages (with ISR for updates).
- **SSR/ISR:** For dynamic pages like product details (to ensure fresh data like price/stock and SEO), user account pages, and potentially search results. ISR is preferred for large product catalogs to balance build times and freshness.[13]
- **CSR:** Client-side rendering might be used within specific components for highly interactive elements or parts of the user dashboard not requiring SEO indexing.[13]

Component Design: Follows a component-based approach, creating reusable UI elements (Buttons, Cards, Modals) and feature-specific components (ProductCard, ProductGrid, CartView).

State Management: Depending on complexity, state management could utilize React's built-in Context API for simpler global state (like authentication status, cart) or a dedicated library like Zustand or Redux for more complex state interactions. Local component state is used where appropriate.

Styling: Tailwind CSS is used exclusively for styling.[19] Utility classes are applied directly in JSX components. The tailwind.config.js file is configured with any custom theme extensions (colors, fonts, spacing). Responsive design is implemented using Tailwind's breakpoint prefixes (sm:, md:, lg:) adhering to mobile-first principles.[20]

Data Fetching:

- `getStaticProps` (with revalidate for ISR) or `getServerSideProps` are used for server-rendering data.[52]
- Axios is used for client-side data fetching (e.g., fetching cart updates, submitting forms) interacting with the Express backend API.[36] Axios interceptors are configured for adding the JWT Authorization header to requests and handling API errors globally.[45]

3.5 Backend Architecture (Express.js)

The backend API is built with Express.js, focusing on modularity, security, and maintainability.

Project Structure: Adopts a layered, modular structure:

- **routes/:** Defines API endpoints, mapping paths and HTTP methods to controllers. Separate files for logical resource groups (e.g., product.routes.js, auth.routes.js, order.routes.js, admin.routes.js).[47]
- **controllers/:** Contains functions that handle incoming requests, process input, interact with services, and formulate responses.[47]
- **services/:** Encapsulates business logic, orchestrating interactions between controllers and the data access layer.[47]
- **models/:** Defines data structures/schemas (e.g., Mongoose schemas if using MongoDB).[49]
- **middleware/:** Houses reusable middleware functions (authentication checks, authorization (RBAC), input validation, logging, error handling).[47]
- **config/:** Stores configuration files (database connection, JWT secrets, etc.).
- **utils/:** Contains utility functions (e.g., password hashing, helper functions).
- **server.js / app.js:** Entry point, initializes Express app, applies global middleware, mounts routers, starts the server.[47]

Middleware Strategy:

- **Application-level:** Used for global concerns like CORS, body parsing (express.json(), express.urlencoded()), request logging (e.g., morgan), basic security headers (helmet), and potentially global error handling.[18]
- **Router-level:** Used for applying middleware to specific route groups (e.g., applying JWT authentication middleware to all /api/v1/orders routes).[27]
- **Route-specific:** Applied directly in route definitions for specific checks (e.g., RBAC middleware checking for 'admin' role on a specific admin endpoint).[50]

Error Handling: A dedicated error-handling middleware function (with err, req, res, next signature) is defined last to catch and format errors consistently.[27]

API Design: Follows RESTful principles for structuring endpoints and using HTTP methods appropriately for CRUD operations.[48] Uses JSON for request and response bodies.[90] Employs standard HTTP status codes to indicate outcomes.[91]

3.6 Database Schema Design

The MongoDB schema design focuses on organizing e-commerce data effectively, balancing query performance and data integrity using NoSQL principles. Key collections include:

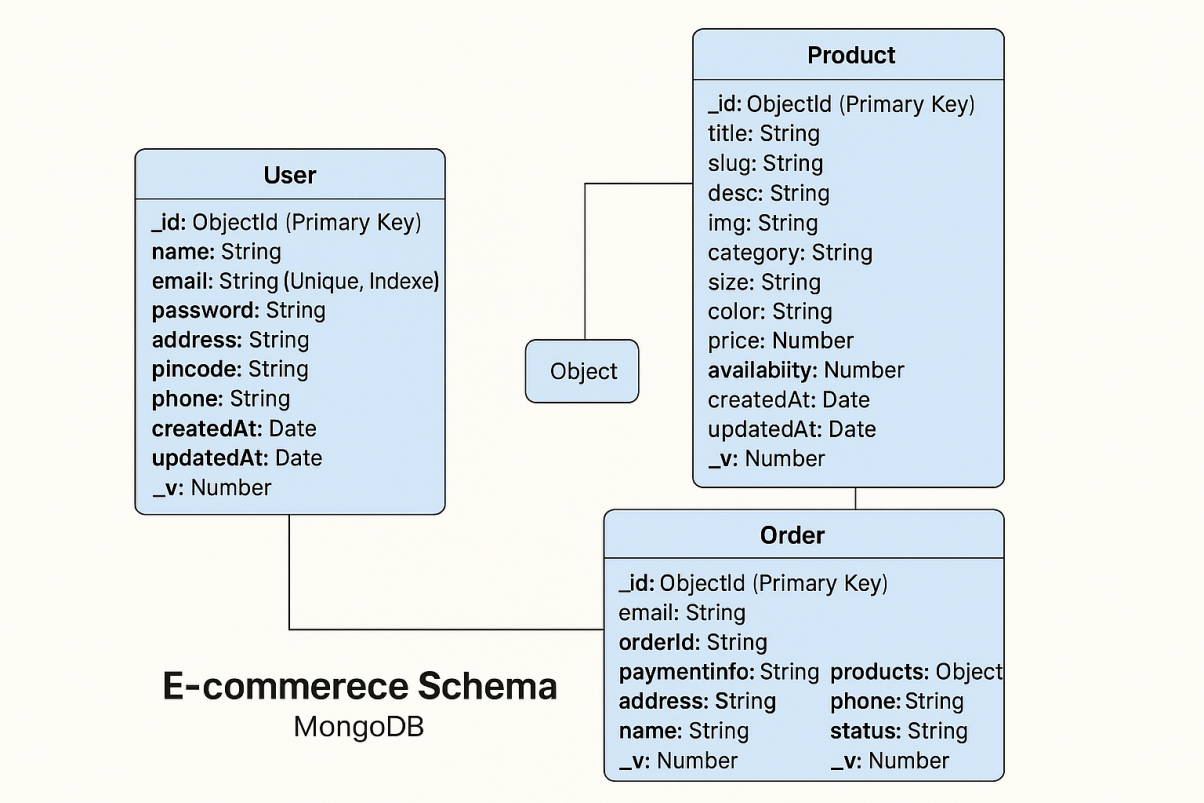


Figure 3.2: E-Commerce Schema

- **users**

SQL

```
_id: ObjectId (Primary Key)
name: String
email: String (Unique, Indexed)
password: String
address: String
pincode: String
phone: String
createdAt: Date
updatedAt: Date
__v: Number
```

- **products**

SQL

```
_id: ObjectId (Primary Key)
title: String
slug: String
desc: String
img: String
category: String
size: String
color: String
price: Number
availablty: Number
createdAt: Date
updatedAt: Date
__v: Number
```

- **orders**

SQL

```
_id: ObjectId (Primary Key)
email: String
orderId: String
paymentInfo: String
products: Object
address: String
city: String
state: String
pincode: String
phone: String
name: String
amount: Number
status: String
deliveryStatus: String
createdAt: Date
updatedAt: Date
__v: Number
```

Design Rationale:

- **Embedding vs. Referencing:** Embedded documents (like addresses within Users, items within Orders) are used where data is closely tied and frequently accessed together, optimizing read performance.[87] References (ObjectIds) are used for many-to-many relationships (Products-Categories) or when entities have independent lifecycles (Users-Orders).
- **Indexing:** Indexes are strategically placed on fields used for querying and sorting (e.g., email, sku, userId, status, category) to ensure efficient data retrieval.[86]
- **Data Types:** Appropriate data types are used (e.g., NumberDecimal for currency to avoid floating-point issues).

- **Consistency:** While NoSQL is flexible, the application layer enforces schema validation (e.g., using Mongoose) to maintain data integrity.[86] Transactional integrity for critical operations like order placement and payment updates needs careful handling at the application level or using MongoDB's multi-document transaction features if required.
- **Product Variants:** The schema needs a clear strategy for variants.[88] Embedding variants within the product document works for a limited number, but a separate Variants collection linked to the main Product might scale better for complex scenarios, allowing individual SKU tracking and pricing.[89] This design assumes a simpler embedded approach initially.

The decision between SQL and NoSQL often involves trade-offs. While SQL databases offer strong consistency and standardized querying, ideal for the transactional nature of user accounts and orders[87], NoSQL databases like MongoDB provide flexibility and potentially better read performance for complex, evolving product catalogs.[85] A hybrid approach, using SQL for transactional data and NoSQL for catalog data, is a common pattern in large e-commerce systems to leverage the best of both worlds.[85] However, for this project, sticking with MongoDB maintains the unified JavaScript/JSON-centric stack, simplifying development, while still requiring careful schema design and application-level logic to manage relationships and consistency.

3.7 API Design (RESTful Principles)

The backend Express.js application exposes a RESTful API for the frontend (and potentially future admin interfaces or third-party integrations) to interact with. The design adheres to standard REST principles for clarity, consistency, and predictability.[91]

Resource-Based URLs: Endpoints are designed around resources (nouns), not actions (verbs). Example: /api/v1/products, /api/v1/products/{productId}, /api/v1/users/{userId}/orders.[49]

HTTP Methods for CRUD:

- **GET:** Retrieve resources (e.g., GET /api/v1/products for list, GET /api/v1/products/{productId} for single).[48]
- **POST:** Create new resources (e.g., POST /api/v1/orders, POST /api/v1/auth/login).[48]
- **PUT:** Update/replace an existing resource entirely (e.g., PUT /api/v1/products/{productId}).[48]
- **PATCH:** Partially update an existing resource (e.g., PATCH /api/v1/orders/{orderId} to update status).[48]
- **DELETE:** Remove a resource (e.g., DELETE /api/v1/products/{productId} - likely admin only).[48]

JSON Data Format: Requests and responses primarily use the JSON format for data exchange. Content-Type: application/json header is used.[54]

Statelessness: Each API request contains all necessary information for the server to process it; the server does not rely on stored session state between requests (facilitated by JWT authentication).

Status Codes: Appropriate HTTP status codes are used to indicate the outcome of requests:

- 2xx (Success): e.g., 200 OK, 201 Created, 204 No Content[54]
- 4xx (Client Error): e.g., 400 Bad Request (validation failed), 401 Unauthorized (not authenticated), 403 Forbidden (not authorized), 404 Not Found[54]
- 5xx (Server Error): e.g., 500 Internal Server Error.[91]

Versioning: API versioning (e.g., /api/v1/) is included in the URL path to allow for future non-breaking changes.

Modularity: Routes are organized modularly within the Express application, corresponding to the resource structure (as detailed in Section 3.5).[47]

This consistent RESTful design makes the API easier to understand, consume, and maintain for both the frontend developers and potentially for future integration.

Chapter 4
Implementation Details

This chapter delves into the practical implementation of the e-commerce platform, translating the architectural design outlined in Chapter 3 into functional code. It covers the development specifics of the frontend using Next.js and Tailwind CSS, the backend API using Express.js, database interactions, the JWT authentication mechanism, Razorpay payment integration, client-server communication via Axios, and the foundational APIs for the admin panel.

4.1 Frontend Development (Next.js & Tailwind CSS)

The frontend aims to provide a fast, responsive, and engaging user experience, built upon Next.js and styled with Tailwind CSS.

- **UI Components:** Key reusable UI components were developed, including:
  - **ProductCard:** Displays product image, name, price, and an "Add to Cart" button. Styled using Tailwind utilities for layout, typography, borders, and hover effects.
  - **ProductGrid:** Arranges `ProductCard` components in a responsive grid using Tailwind's grid and gap utilities (`grid`, `grid-cols-1`, `sm:grid-cols-2`, `md:grid-cols-3`, `lg:grid-cols-4`, `gap-6`).[55]
  - **ProductDetail:** Displays comprehensive product information, including image gallery, description, price, variant selectors (if applicable), and "Add to Cart" button. Layout structured using Flexbox or Grid utilities.
  - **Header/Navbar:** Contains branding, navigation links (categories), search bar (optional), cart icon, and user authentication status/login link. Implemented with responsive variants for mobile (hamburger menu) and desktop views.[92]
  - **Footer:** Standard footer content (links, copyright).
  - **CartView:** Displays items in the cart, quantities, total price, and proceeds to checkout button. Often implemented as a sidebar or modal.
  - **CheckoutForm:** Multi-step or single-page form for collecting shipping address, billing address (optional), and triggering payment.
- **Styling (Tailwind CSS):** Tailwind's utility classes were applied directly within JSX elements. For example, a button might be styled as:

JavaScript

```
<button␣className="bg-blue-500␣hover:bg-blue-700␣
   text-white␣font-bold␣py-2␣px-4␣rounded">...</
   button>
```

[21]

Responsive design was achieved using prefixes:

JavaScript

```
<div␣className="w-full␣md:w-1/2␣lg:w-1/3␣p-4">...</
   div>
```

[20]

A mobile-first approach was generally followed, defining base styles for mobile and overriding for larger screens [21]. The `tailwind.config.js` file was used to define any custom theme colors or extensions.

- **Routing:** Next.js's file-based routing was used. Static routes like `/about` were created by `pages/about.js` [26]. Dynamic routes for products (`/products/[productId]`) and categories (`/category/[slug]`) utilized bracket notation to capture parameters from the URL [26]. The `next/link` component was used for client-side navigation between pages, enabling prefetching for faster transitions [53].
- **Data Fetching:**
  - `getStaticProps`: Used for pages like `/about` or potentially category pages, fetching data at build time. The `revalidate` option was added for ISR where periodic updates were needed (e.g., category pages).[52]
  - `getServerSideProps`: Used for pages requiring fresh data on every request, such as the main product detail page (`/products/[productId]`) to fetch current price and stock, or user-specific pages like `/account/orders`.[13]
  - **Client-Side Fetching (Axios):** Used within components (inside `useEffect` or triggered by user actions) to fetch dynamic data like cart contents, check authentication status, or submit forms to the Express backend API.[36]
- **State Management:**
  - The React Context API was employed for managing global state such as user authentication status and shopping cart contents.
  - This provided a simpler alternative to Redux for this project's scope.
  - Local component state (`useState`, `useReducer`) managed UI-specific state.

4.2 Backend Development (Express.js)

The Express.js backend provides the core API logic, database interaction, and security enforcement.

- **Modular Structure:** The backend codebase was organized following the layered architecture described in Chapter 3.
- **Routes** (`routes/`)**:**
  - Files like `product.routes.js`, `auth.routes.js`, `order.routes.js`, `admin.routes.js` define endpoints using `express.Router()`.
  - They import controllers and apply route-specific middleware (like authentication or RBAC checks).[47]
  - **Example** (`product.routes.js`)**:**

JavaScript

```
const express = require('express');
const productController = require('../
    controllers/product.controller');
const authMiddleware = require('../middleware/
    auth.middleware'); // JWT verification
const adminMiddleware = require('../middleware/
    admin.middleware'); // RBAC check

const router = express.Router();

router.get('/', productController.
    getAllProducts);
router.get('/:id', productController.
    getProductById);

// Admin-only routes
router.post('/', authMiddleware,
    adminMiddleware, productController.
    createProduct);
router.put('/:id', authMiddleware,
    adminMiddleware, productController.
    updateProduct);
router.delete('/:id', authMiddleware,
    adminMiddleware, productController.
    deleteProduct);

module.exports = router;
```

- **Controllers** (`controllers/`)**:**
  - Functions handle `req` and `res` objects, extract data (from `req.body`, `req.params`, `req.query`).

- Call corresponding service functions and send responses (e.g., `res.json()`, `res.status().send()`).[47]
- **Example (`product.controller.js`):**

JavaScript

```
const productService = require('../services/
    product.service');

exports.getAllProducts = async (req, res, next)
    => {
    try {
        const products = await productService.
            findAllProducts(req.query); // Pass
            query params for filtering/
            pagination
        res.status(200).json(products);
    } catch (error) {
        next(error); // Pass error to error
            handling middleware
    }
};
//... other controller methods (getProductById,
    createProduct, etc.)
```

- **Services (`services/`):**
  - Contain the core business logic, interacting with models/database functions.[47]
  - **Example (`product.service.js`):**

JavaScript

```
const Product = require('../models/product.
    model'); // Assuming Mongoose model

exports.findAllProducts = async (queryParams)
    => {
    // Add logic for filtering, sorting,
        pagination based on queryParams
    return await Product.find({ isActive: true
        /*... other filters */ });
};
//... other service methods (findProductById,
    addProduct, etc.)
```

- **Middleware (`middleware/`):** Includes functions for:
  - `auth.middleware.js`: Verifies JWT token from Authorization header.[66]

- `admin.middleware.js` / `rbac.middleware.js`: Checks user role (extracted from verified JWT payload) against required roles for the route.[50]
- `validation.middleware.js`: Uses libraries like `express-validator` to validate request bodies/params.[18]
- `error.handler.js`: Centralized error handling logic.[27]

- **CRUD Operations:[48]**
  - Standard CRUD operations were implemented for key resources like Products, Users, and Orders.
  - Products were managed via admin API.
  - Users included registration and profile updates.
  - Orders included creation and retrieval.
  - HTTP methods followed REST principles:
    - `POST` for creation,
    - `GET` for retrieval,
    - `PUT/PATCH` for updates,
    - `DELETE` for removal.

4.3 Database Implementation

- **Connection:[49]**
  - A connection to the MongoDB database was established using Mongoose in the main application file (`server.js`) or a dedicated configuration file.
  - Connection strings were stored securely in environment variables.
- **Models:[49]**
  - Mongoose schemas were defined in the `models/` directory for each collection, including `User`, `Product`, `Order`, and `Category`, as outlined in Chapter 3.6.
  - These schemas defined field types, validation rules, default values, and indexes.
  - **Example (`models/product.model.js`):**

JavaScript

```
const mongoose = require('mongoose');
const { Schema } = mongoose;

const productSchema = new Schema({
    title: { type: String, required: true },
    slug: { type: String, required: true,
       unique: true },
```

```
        desc: { type: String },
        img: { type: String },
        category: { type: String, index: true },
        size: { type: String },
        color: { type: String },
        price: { type: Number, required: true },
        availability: { type: Number, required:
           true, min: 0 },
        isActive: { type: Boolean, default: true }
    }, { timestamps: true });

    module.exports = mongoose.model('Product',
       productSchema);
```

- **Data Access Logic:[49]**
  - All database interactions (queries, updates) were encapsulated within the service layer or a dedicated data access layer.
  - Mongoose methods were used such as `.find()`, `.findById()`, `.create()`, `.updateOne()`, and `.deleteOne()`.
  - Asynchronous operations were handled using `async/await` to ensure proper flow control and error handling.

4.4 Authentication Flow Implementation (JWT)

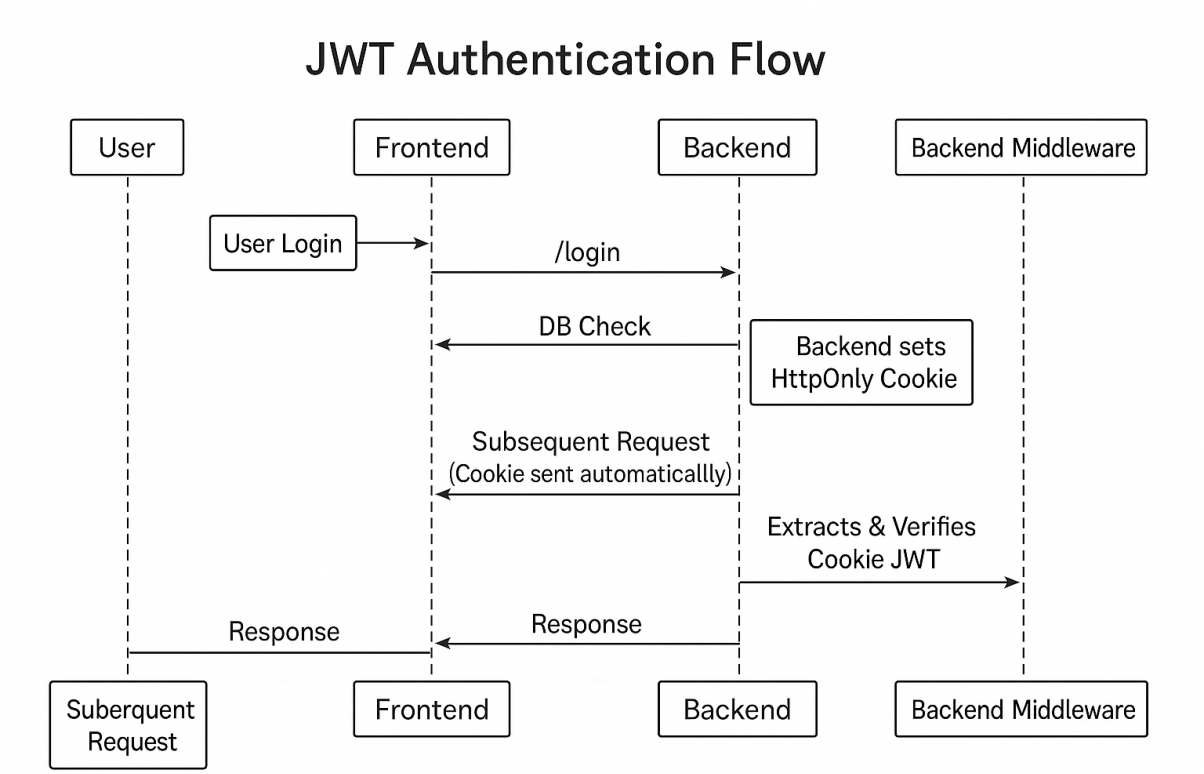


Figure 4.1: JWT authentication flow

The JWT authentication flow was implemented as follows:

- **User Registration/Login:**
  - **Frontend:**
    - A form (`<form>`) collects credentials (e.g., email, password).
    - On submit, an Axios `POST` request is sent to the backend endpoint[70] `/api/v1/auth/register` or `/api/v1/auth/login`.
  - **Backend Verification:**
    - The Express controller receives the request.
    - For registration: Validates input, hashes the password using `bcrypt`[35], and creates a new user in the database.[50]
    - For login: Finds the user by email/username, compares the submitted password with the stored hash using `bcrypt.compare()`.[67]
- **JWT Generation:**
  - If credentials are valid (login) or registration is successful, the backend generates a JWT using the `jsonwebtoken` library.[65]

- The payload includes essential, non-sensitive user identifiers (e.g., `userId`, `roles`).[50] Sensitive data is never included.[39]
- A secret key (stored in environment variables) is used for signing (e.g., HS256 algorithm).[65]
- An expiration time is set (e.g., `'1h'` for access tokens).[16]

JavaScript

```
// Example JWT Generation in auth.controller.js
const jwt = require('jsonwebtoken');
//... after successful login/registration
const payload = { userId: user._id, roles: user.
   roles };
const secret = process.env.JWT_SECRET;
const options = { expiresIn: '1h' }; // Token
   expires in 1 hour
const token = jwt.sign(payload, secret, options);
```

- **Token Transmission:**
  - The generated JWT is sent back to the client in the response body.[67]
  - Alternatively, and more securely, it can be set as an `HttpOnly`, `Secure` cookie.[16]
  - This project implements the HttpOnly cookie method.

JavaScript

```
// Example setting HttpOnly cookie in auth.
   controller.js
res.cookie('jwtToken', token, {
    httpOnly: true, // Prevents client-side JS
       access
    secure: process.env.NODE_ENV === 'production',
       // Only send over HTTPS in production
    sameSite: 'strict', // Mitigate CSRF
    maxAge: 3600000 // 1 hour expiry (in
       milliseconds)
});
res.status(200).json({ success: true, userId: user.
   _id /* other non-sensitive data */ });
```

- **Client Storage:**
  - The browser automatically stores the HttpOnly cookie.[16]
  - No client-side JavaScript storage (e.g., `localStorage`) is used for the token itself.
- **Subsequent Requests:**

- For requests to protected frontend routes or backend API endpoints, the browser automatically includes the `jwtToken` cookie.
- If not using cookies, the client must manually retrieve the token and add it to the `Authorization: Bearer <token>` header via an Axios request interceptor.[16]

- **Backend Verification (Middleware):**
  - An Express middleware function (`auth.middleware.js`) intercepts requests to protected routes.[66]
  - It extracts the token (from cookie or Authorization header).
  - Verifies token signature and expiration using `jwt.verify()` with the secret key.[65]
  - If valid, decodes the payload, attaches user info (like `userId` and `roles`) to `req.user`, and calls `next()`.[65]
  - If invalid (missing, expired, tampered), it sends a `401 Unauthorized` or `403 Forbidden` response.[67]

JavaScript

```
// Example auth.middleware.js
const jwt = require('jsonwebtoken');

module.exports = (req, res, next) => {
    try {
        const token = req.cookies.jwtToken; // Extract from HttpOnly cookie
        if (!token) {
            throw new Error('Authentication_failed!');
        }
        const decodedToken = jwt.verify(token, process.env.JWT_SECRET);
        req.user = { userId: decodedToken.userId, roles: decodedToken.roles };
        next();
    } catch (err) {
        res.status(401).json({ message: 'Authentication_failed!' });
    }
};
```

4.5 Payment Gateway Integration (Razorpay)

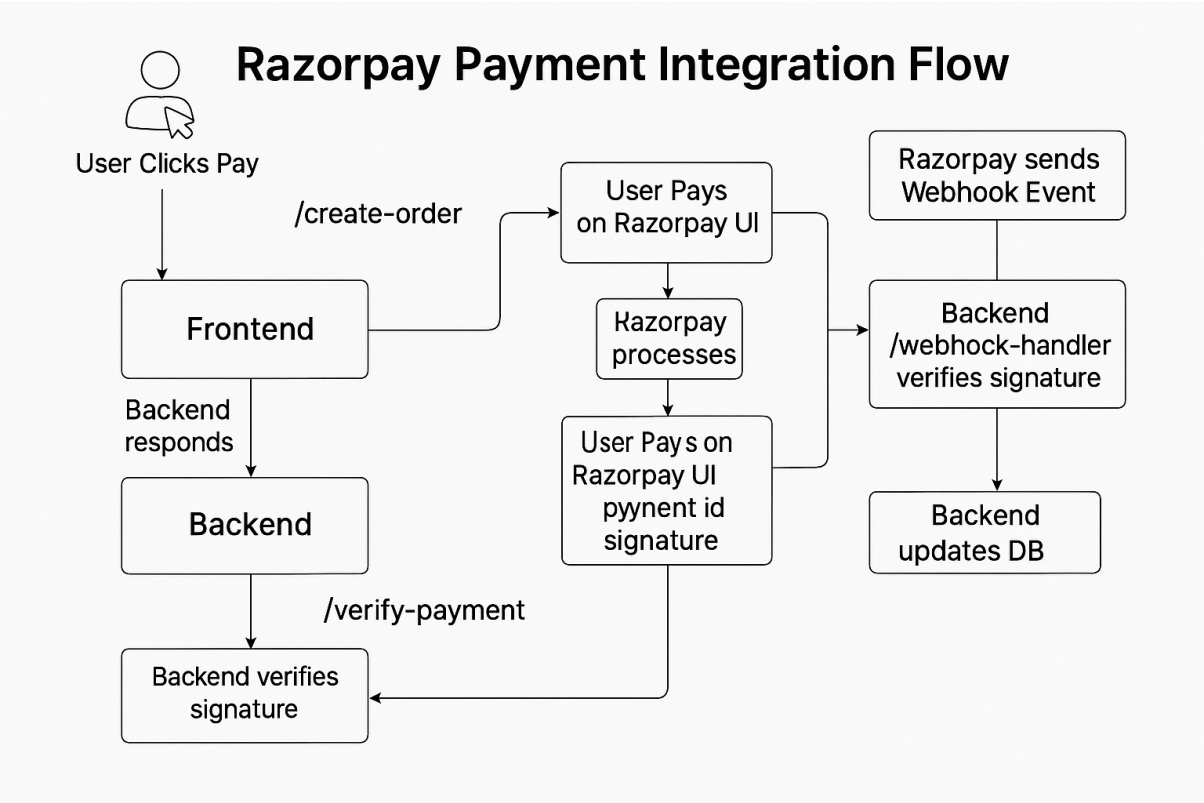


Figure 4.2: Razorpay Integration Flow

Integrating Razorpay involved frontend checkout initiation and backend order creation and verification.

- **Order Creation (Backend):**
  When the user proceeds to checkout on the frontend, the frontend sends the cart total amount to a dedicated backend endpoint (e.g., `POST /api/v1/payments/create-order`). The Express controller for this route uses the Razorpay Node.js SDK to call the Razorpay Orders API. Required parameters include:[76]
  - `amount` (in the smallest currency unit, e.g., paise for INR),[96]
  - `currency` (e.g., 'INR'),
  - a unique `receipt` ID.

  Optional notes can also be added.

JavaScript

```
// Example in payment.controller.js (using Razorpay Node.js SDK)
const Razorpay = require('razorpay');
const instance = new Razorpay({
```

```
        key_id: process.env.RAZORPAY_KEY_ID,
        key_secret: process.env.RAZORPAY_KEY_SECRET,
    });

    exports.createOrder = async (req, res, next) => {
        try {
            const options = {
                amount: req.body.amount * 100, // Amount in paise
                currency: "INR",
                receipt: `receipt_order_${Date.now()}`, // Generate a unique receipt ID
                payment_capture: 1 // Auto capture payment
            };
            const order = await instance.orders.create(options);
            if (!order) return res.status(500).send("Error creating order");
            res.json(order); // Send order details (including order_id) back to frontend
        } catch (error) {
            next(error);
        }
    };
```

- **Checkout Initiation (Frontend):**
  - The frontend receives the `order_id` from the backend response.
  - It includes the Razorpay Checkout script:[77]

    ```
    <script src="https://checkout.razorpay.com/v1/checkout.js"></script>
    ```

  - It configures the Razorpay Checkout `options` object, including:
    - `key` (Razorpay Key ID),
    - `amount`, `currency`,
    - `name`, `description`,
    - the received `order_id`, and
    - a `handler` function (or `callback_url`).
  - `prefill` options can be used to enhance user experience.[95]
  - It creates a new `Razorpay(options)` instance and calls `.open()` to launch the payment modal.[76]

JavaScript

```javascript
// Example frontend JavaScript
const handlePayment = async (orderAmount) => {
    // 1. Create order on backend
    const orderResponse = await fetch('/api/v1/payments/create-order', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json', /* + Auth header if needed */ },
        body: JSON.stringify({ amount: orderAmount })
    });
    const order = await orderResponse.json();

    // 2. Configure and open Razorpay Checkout
    const options = {
        key: "YOUR_RAZORPAY_KEY_ID", // From backend or env
        amount: order.amount, // Amount from order response (in paise)
        currency: order.currency,
        name: "Your E-commerce Site",
        description: "Test Transaction",
        order_id: order.id, // Pass the order ID received from backend
        handler: function (response) {
            // 3. Handle success: Send response details to backend for verification
            verifyPaymentOnBackend(response);
        },
        prefill: {
            name: "Customer Name",
            email: "customer@example.com",
            contact: "9999999999"
        },
        theme: {
            color: "#3399cc"
        }
    };
    const rzp = new Razorpay(options);
    rzp.on('payment.failed', function (response){
        // Handle payment failure on client side
        alert("Payment Failed: " + response.error.description);
    });
    rzp.open();
};
```

- **Payment Handling & Signature Verification (Backend):**
  - On successful payment via the Razorpay modal, the `handler` function (or `callback_url`) receives a response containing:[74]
    - `razorpay_payment_id`,
    - `razorpay_order_id`, and
    - `razorpay_signature`.
  - The frontend sends these details to a dedicated backend verification endpoint (e.g., `POST /api/v1/payments/verify`).
  - The backend must verify the `razorpay_signature` to confirm authenticity and prevent tampering.[74]
  - Verification involves generating an HMAC-SHA256 hash using the `order_id` and the `razorpay_payment_id`, using the Razorpay Key Secret.[78]
  - This generated signature is compared with the received `razorpay_signature`. If they match, the payment is considered authentic.[78]

JavaScript

```
// Example in payment.controller.js (verify endpoint)
const crypto = require('crypto');

exports.verifyPayment = (req, res, next) => {
    const { order_id, razorpay_payment_id, razorpay_signature } = req.body;
    const secret = process.env.RAZORPAY_KEY_SECRET;

    // IMPORTANT: Fetch the original order details (especially amount) from your DB
    // using the order_id received from YOUR system before checkout, not razorpay_order_id from client.
    // Let's assume original_order_id was stored and retrieved.

    const generated_signature = crypto.createHmac('sha256', secret)
                                      .update(order_id + "|" + razorpay_payment_id) // Use the correct order_id from your system
                                      .digest('hex');
```

```
15
16          if (generated_signature === razorpay_signature)
                {
17              // Payment is authentic. Process order (
                    update status in DB, etc.)
18              console.log("Payment␣is␣successful␣and␣
                    verified");
19              // Redirect user to success page or send
                    success response
20              res.status(200).json({ success: true,
                    message: "Payment␣verified␣successfully"
                     });
21              // Ideally, final confirmation relies on
                    webhook
22          } else {
23              res.status(400).json({ success: false,
                    message: "Payment␣verification␣failed"
                    });
24          }
25      };
```

- **Webhooks (Backend):**
  - For robust, asynchronous confirmation of events (e.g., `payment.captured`, `payment.failed`, `order.paid`, `refunds`), Razorpay Webhooks are essential.[77]
  - A dedicated backend endpoint (e.g., `POST /api/v1/payments/webhook`) is created to receive webhook notifications.[80]
  - This endpoint must be configured in the Razorpay dashboard along with a secret.
  - The webhook handler must verify the incoming request's signature (from the[79] `X-Razorpay-Signature` header) using the webhook payload and the secret.
  - Upon verification, the handler processes the event (e.g., updating order status in the database).
  - Implementing idempotency (checking event IDs) is important to prevent duplicate processing.[79]

JavaScript

```
1       // Example Express webhook handler
2       app.post('/api/v1/payments/webhook', express.raw({
            type: 'application/json' }), (req, res) => {
3           const secret = process.env.
                RAZORPAY_WEBHOOK_SECRET;
4           const signature = req.headers['x-razorpay-
                signature'];
5
```

```
        try {
            const shasum = crypto.createHmac('sha256', secret);
            shasum.update(req.body); // Use raw body
            const digest = shasum.digest('hex');

            if (digest === signature) {
                console.log('Webhook verified successfully');
                const event = JSON.parse(req.body.toString()); // Parse raw body after verification

                // Handle the event (e.g., payment.captured, order.paid)
                if (event.event === 'payment.captured') {
                    const paymentEntity = event.payload.payment.entity;
                    console.log(`Payment ${paymentEntity.id} captured for Order ${paymentEntity.order_id}`);
                    // Update order status in database
                } else if (event.event === 'order.paid') {
                    const orderEntity = event.payload.order.entity;
                    console.log(`Order ${orderEntity.id} paid`);
                    // Update order status if needed
                }
                //... handle other events

                res.status(200).send('OK');
            } else {
                console.log('Invalid webhook signature');
                res.status(400).send('Invalid Signature');
            }
        } catch (error) {
            console.error('Error processing webhook:', error);
            res.status(500).send('Webhook processing error');
        }
    });
```

4.6 Client-Server Communication (Axios)

Axios was used for all HTTP communication between the Next.js frontend and the Express.js backend.

- **API Requests:** Standard Axios methods (`axios.get`, `axios.post`, `axios.put`, `axios.delete`) were used within frontend components or utility functions to interact with the backend API endpoints.
- **Axios Instance:** A centralized Axios instance was created (`utils/axiosInstance.js`) with base URL configuration and default settings.[36]

JavaScript

```
// utils/axiosInstance.js
import axios from 'axios';

const axiosInstance = axios.create({
    baseURL: process.env.NEXT_PUBLIC_API_BASE_URL
       ||
            'http://localhost:5000/api/v1', //
               Example
    timeout: 10000, // 10 second timeout
    // Cookies should be handled automatically by
       the browser if backend sets HttpOnly
    // If using Authorization header:
    // headers: { 'Content-Type': 'application/json
       ' }
});

// Add interceptors (see below)

export default axiosInstance;
```

- **Request Interceptor:** An interceptor was added to the instance to modify outgoing requests. If using header-based JWT authentication (instead of cookies), this interceptor would retrieve the token from secure storage (not `localStorage`) and add the `Authorization: Bearer <token>` header. Since HttpOnly cookies are used in this implementation, this interceptor might primarily be used for adding other common headers if needed.[36]

JavaScript

```
// In axiosInstance.js (Example if using header
   auth)
axiosInstance.interceptors.request.use(
    config => {
        const token = getAuthToken(); // Function
           to get token from secure storage
```

```
            if (token) {
                config.headers['Authorization'] = `
                    Bearer ${token}`;
            }
            return config;
        },
        error => Promise.reject(error)
    );
```

- **Response Interceptor:** A response interceptor was implemented for global error handling. It checks the status code of error responses and takes appropriate action, such as redirecting to login on 401 Unauthorized, showing generic error messages for 500 Internal Server Error, or handling network errors. This centralizes error handling logic, keeping component code cleaner.[36]

JavaScript

```
    // In axiosInstance.js
    axiosInstance.interceptors.response.use(
        response => response, // Pass through
            successful responses
        error => {
            if (error.response) {
                // Handle specific status codes
                if (error.response.status === 401) {
                    // Handle unauthorized access, e.g
                        ., redirect to login
                    console.error("Unauthorized access
                        - 401");
                    // window.location.href = '/login';
                        // Or use Next.js router
                } else if (error.response.status ===
                    403) {
                    console.error("Forbidden access -
                        403");
                    // Show forbidden message
                } else if (error.response.status ===
                    404) {
                    console.error("Resource not found
                        - 404");
                    // Show not found message
                } else if (error.response.status >=
                    500) {
                    console.error("Server error - 5xx")
                        ;
                    // Show generic server error
                        message
                }
```

```
            } else if (error.request) {
                // Handle network errors (no response
                    received)
                console.error("Network error:", error.
                    request);
                 // Show network error message
            } else {
                // Handle other errors
                console.error("Axios error:", error.
                    message);
            }
            // Return a rejected promise to allow
                component-level catch blocks if needed
            return Promise.reject(error);
        }
    );
```

4.7 Admin Panel API Implementation

To support future administrative functionalities (Objective #11), dedicated backend API endpoints were created within the Express.js application.

- **Structure:**
  - Admin-specific routes were grouped under a distinct prefix, typically `/api/v1/admin/`, and managed in separate route files such as:
    - `routes/admin/product.routes.js`
    - `routes/admin/order.routes.js`
    - `routes/admin/user.routes.js`
- **Functionality:** These routes implement CRUD operations for managing core e-commerce entities:
  - **Product Management**[48]
    - `POST /admin/products` — Create new product
    - `GET /admin/products` — Retrieve all products
    - `GET /admin/products/{id}` — Retrieve product by ID
    - `PUT /admin/products/{id}` — Update product
    - `DELETE /admin/products/{id}` — Delete product
  - **Order Management**
    - `GET /admin/orders` — View all orders
    - `GET /admin/orders/{id}` — View specific order details
    - `PATCH /admin/orders/{id}` — Update order status

  - **User Management**
    * `GET /admin/users` — View all users
    * `GET /admin/users/{id}` — View specific user details
    * `PATCH /admin/users/{id}` — Update user roles/status
- **Protection:**
  - All admin endpoints are secured using:
    * `auth.middleware.js` — Verifies JWT token and extracts user info.
    * `admin.middleware.js` or RBAC middleware — Ensures that the user has an `'ADMIN'` role by checking `req.user.roles`.

This modular and secure design forms a reliable foundation for building a separate admin frontend interface, enabling authorized administrators to manage the platform's data effectively.

Chapter 5
Security Implementation and Analysis

Security was a primary objective of this project (Objective #2). This chapter details the implementation and analysis of various security measures, focusing on authentication using JWT, data encryption with AES, secure coding practices aligned with OWASP recommendations, payment security through Razorpay, and authorization via Role-Based Access Control (RBAC).

5.1 Authentication Security

A robust JWT-based authentication system was implemented, adhering to security best practices to mitigate common vulnerabilities.

5.1.1 JWT Implementation Analysis

The implementation involved user login, JWT generation, secure client-side handling (via HttpOnly cookies), transmission, and server-side verification using middleware, as detailed in Section 4.4.

5.1.2 Algorithm Selection

The HS256 (HMAC with SHA-256) algorithm was chosen for signing JWTs. This symmetric algorithm requires a single secret key shared between the signing (server) and verifying (server) parties. While asymmetric algorithms like RS256 offer advantages when verification needs to happen by third parties without sharing the secret, HS256 is simpler and sufficient for this architecture where the same backend server both issues and verifies the tokens[68]. Critically, the implementation explicitly rejects tokens attempting to use the `alg: "none"` header, preventing attackers from bypassing signature verification entirely[16]. Case-insensitive checks (e.g., blocking ”NoNe”) are also implemented[68].

5.1.3 Signature Verification

On every request to a protected backend endpoint, the `auth.middleware.js` rigorously verifies the signature of the incoming JWT using the `jsonwebtoken` library's `verify()` method and the securely stored `JWT_SECRET`[39]. This ensures the token has not been tampered with since it was issued by the server[16]. Failure to verify the signature immediately results in a 401 Unauthorized response[93].

5.1.4 Claim Validation

Beyond signature verification, the `jwt.verify()` function also automatically validates standard registered claims present in the token payload[16]:

- **Expiration (exp):** Tokens are issued with a relatively short expiration time (e.g., 1 hour)[16]. The verification process automatically rejects expired tokens, limiting the window of opportunity for attackers if a token is compromised[69].
- **Issuer (iss) and Audience (aud):** While not explicitly detailed in the implementation snippets, best practice dictates including and validating `iss` (identifying the token issuer, i.e., our backend) and `aud` (identifying the intended recipient, i.e., our API) claims to prevent token misuse across different services[16]. The implementation should be configured to validate these claims if they are included during token generation.

5.1.5 Secure Token Storage

To mitigate the significant risk of token theft via Cross-Site Scripting (XSS) attacks associated with storing JWTs in browser `localStorage` or `sessionStorage`, this project stores the JWT in an HttpOnly cookie[16]. The HttpOnly flag prevents client-side JavaScript from accessing the cookie, making it inaccessible to XSS scripts. Additionally, the Secure flag is set (in production) to ensure the cookie is only sent over HTTPS connections, and the `SameSite=Strict` flag is used to mitigate Cross-Site Request Forgery (CSRF) risks[73].

5.1.6 Token Revocation Strategy

Stateless JWTs cannot be easily invalidated before their expiration time[16]. To handle scenarios like user logout or suspected token compromise, a robust strategy is needed. While not explicitly detailed in the provided implementation snippets, common approaches compatible with this architecture include:

- **Short Expiration + Refresh Tokens:** Using short-lived access tokens (e.g., 1 hour) alongside longer-lived refresh tokens (stored securely, potentially in the database linked to the user session). When the access token expires, the client uses the refresh token (sent via a secure endpoint) to obtain a new access token. Logging out involves invalidating the refresh token on the server-side[16].
- **Token Blacklisting:** Maintaining a server-side list (e.g., in Redis or a database) of invalidated token identifiers (e.g., the `jti` claim). The authentication middleware checks this blacklist after verifying the token signature and claims[16].

This project would ideally implement the refresh token strategy for a balance between security and user experience, invalidating the refresh token on logout.

5.1.7 Key Management

The security of the HS256 algorithm relies entirely on the secrecy and strength of the `JWT_SECRET` key[39]. This key is:

- **Generated Securely:** A long, random, high-entropy string generated using cryptographically secure methods.

- **Stored Securely:** Stored as an environment variable (`process.env.JWT_SECRET`) on the server, never hardcoded in the source code or committed to version control[35]. Access is restricted.
- **Rotation:** While not automated in this implementation, best practices recommend periodic key rotation, especially in high-security environments or if a compromise is suspected[35].

5.1.8 OWASP Broken Authentication Mitigation

The implemented JWT strategy directly addresses several aspects of OWASP Top 10 A07:2021 - Identification and Authentication Failures (previously A2:2017 - Broken Authentication)[62]:

- **Credential Protection:** Passwords are never stored in plaintext; strong hashing (bcrypt) is used[35]. JWTs avoid transmitting passwords after initial login.
- **Session Management:** Using JWTs with short expirations and secure HttpOnly cookies provides more robust session management than traditional server-side sessions susceptible to fixation or hijacking if not configured properly[94]. The rejection of `alg: "none"` and mandatory signature verification prevent token tampering[64]. Implementing a revocation strategy (refresh tokens/blacklist) further strengthens protection.

Chapter 6
Deployment

The application was deployed on Vercel, a cloud platform that offers optimized hosting for Next.js and frontend applications. Vercel is well-suited for projects requiring fast serverless functions, seamless CI/CD, and automatic scaling.

6.1 Why Vercel?

- **Seamless Integration:** Vercel provides built-in support for Next.js, enabling effortless deployment with minimal configuration.
- **Automatic CI/CD:** Once connected to GitHub, Vercel automatically triggers builds and deployments with every push to the repository, ensuring continuous integration and delivery.
- **Global CDN:** The platform automatically distributes content across a global content delivery network, resulting in faster load times and improved performance worldwide.
- **Preview Deployments:** Vercel offers preview deployments for each pull request, allowing testing in a staging environment before final deployment.

6.2 Deployment Process

- **Repository Connection:** The GitHub repository containing the project's code was linked directly to Vercel. This integration ensures that every time new code is pushed, Vercel automatically handles deployment.
- **Environment Variables:** Secure environment variables such as JWT secrets, AES encryption keys, and Razorpay credentials were configured via Vercel's environment settings, ensuring sensitive data is protected and only accessible in production.
- **Automatic Builds:** Vercel's CI/CD pipeline handles building the project every time new code is pushed, ensuring the most up-to-date version is always deployed. Build logs are easily accessible for debugging if needed.
- **Custom Domain and HTTPS:** A custom domain was set up for the project, and Vercel automatically provided SSL certificates, ensuring secure connections via HTTPS. This is critical for maintaining the security of user data, especially for payment processing.

6.3 Performance Benefits

- **Edge Caching:** Vercel's edge caching technology ensures that static assets (images, CSS, JavaScript) are cached at edge locations closer to the user, resulting in reduced load times, even for users on slower connections.
- **Serverless Functions:** For the backend APIs and payment integrations, Vercel uses serverless functions, which scale automatically based on demand. This ensures that the application remains fast and responsive, even with varying levels of traffic.
- **Fast Load Time:** By optimizing static rendering (SSG) and utilizing Vercel's CDN, the application experiences faster load times, reducing latency across all devices.

6.4 Deployment Outcome

- **Public Access:** The application is now publicly accessible via the deployed URL, allowing users to interact with the platform, make payments, and browse products.
- **Real-World Testing:** Deploying on Vercel allowed the project to be tested in a live environment, validating the full user experience — including payment transactions, authentication, and UI responsiveness.
- **Scalable Infrastructure:** With Vercel's serverless model and automatic scaling, the platform can easily handle more users and traffic, making it well-suited for future growth and potential commercial use.

Chapter 7
Conclusion

This project represents a successful attempt to build a modern, lightweight, and secure e-commerce web application using widely adopted technologies such as Next.js, Express.js, Razorpay, JWT, and AES encryption. The goal was to create a responsive, fast-loading application that functions well across all devices—including under low-bandwidth conditions—while ensuring data security and user-friendly design. The platform has been developed with clean architecture, optimized UI, secure authentication, and seamless payment integration, and has been deployed on Vercel for real-world accessibility.

7.1 Conclusions of the Present Work

- **Secure full-stack development achieved:** The project integrates a Next.js frontend and an Express.js backend, delivering a modular, efficient, and scalable application structure.
- **Authentication and data protection implemented:** JWT-based login and AES encryption help protect user data and sessions, ensuring secure access control.
- **Responsive and optimized UI:** Using Tailwind CSS and React Icons, the UI is mobile-first, lightweight, and visually appealing, performing smoothly across devices.
- **Efficient API communication:** Axios ensures seamless, asynchronous communication between frontend and backend for data fetching and user interactions.
- **Real-time payment integration:** Razorpay integration allows users to make live, secure payments through a trusted payment gateway.
- **Deployment to production environment:** The complete system has been deployed on Vercel, enabling public access with reliable hosting and fast performance.

7.2 Limitations of the Present Work

- **No admin dashboard:** The current system lacks an administrative panel for managing product inventory, user data, and orders. This limits backend control and scalability for store owners.
- **Basic payment flow only:** While Razorpay integration exists, there is no support for features like refund handling, transaction history, or multi-gateway options.
- **No real-time stock management:** Product quantities are static; the application does not update inventory based on purchases or stock limits.

- **Limited user features:** There are no user-centric enhancements like wishlists, product reviews, or ratings, which are essential in competitive e-commerce environments.
- **No search or filtering system:** Users cannot filter products by category, price, or tags, which affects usability and product discoverability.
- **No notification system:** The app does not provide email or SMS alerts for order confirmation, shipping updates, or password resets.
- **Not containerized:** The backend runs on a standard server setup and is not yet containerized using tools like Docker, limiting flexibility and scalability.
- **No database integration in production:** The system does not currently connect to a robust database like MongoDB or PostgreSQL for persistent data storage and querying.
- **Accessibility and localization not addressed:** The UI has not been tested for accessibility (e.g., screen readers) or international language support, which are essential for global reach.

7.3 Scope for Future Work

- **Admin panel development:** Add a secure, role-based admin dashboard with full CRUD operations for managing users, orders, and inventory.
- **Advanced payment features:** Implement refund processing, multiple payment gateway options, and a transaction history system.
- **Real-time inventory tracking:** Introduce automatic stock updates based on orders and notify admin of low inventory thresholds.
- **Search and filter integration:** Build product search, filtering by price, category, and relevance to improve shopping experience.
- **User personalization features:** Add user wishlists, saved carts, reviews, and product recommendations using analytics or machine learning.
- **Notification system:** Integrate email and SMS services (like SendGrid or Twilio) to notify users about orders, account changes, and promotions.
- **Containerization and scaling:** Use Docker for containerization and Kubernetes for orchestration to prepare for horizontal scaling and cloud deployment.
- **Database optimization:** Connect the app to a production-ready database (e.g., MongoDB, PostgreSQL) and implement indexing, caching, and backup strategies.
- **Accessibility and localization:** Conduct WCAG-compliant accessibility testing and introduce multilingual support to make the platform inclusive and global-ready.

References

[1] Pimberly, “10 Challenges for eCommerce in 2025”.

[2] Cloudflare, “How website performance affects conversion rates”.

[3] Azion, “Why Website Performance Impact Conversion Rates?”.

[4] ResultFirst, “How Site Speed Impacts Conversion Rate of Ecommerce Website”.

[5] Ampcus Cyber, “PCI DSS Compliance for U.S. E-Commerce: A 2025 Guide”.

[6] TestingXperts, “Addressing Digital Retail Challenges with PCI DSS Compliance”.

[7] UpGuard, “The Role of Cybersecurity in Protecting E-Commerce Companies”.

[8] Secureframe, “Benefits of PCI DSS Compliance: 4 Reasons Your Business Needs to Comply”.

[9] Speed Commerce, “Why It’s Important to be PCI Compliant?”.

[10] Payments CMI, “India’s E-Commerce Market Growth: Trends & Key Data 2024–2025”.

[11] Shopify, “Global Ecommerce Sales Growth Report [Updated Oct 2024]”.

[12] Convertcart, “eCommerce Checkout UX: 13 Tips To Boost Conversions (+ Templates)”.

[13] Next.js, “SEO: Rendering Strategies”.

[14] Vercel, “Next.js on Vercel”.

[15] August Infotech, “Next.js Performance Optimization: Best Practices for Faster Apps”.

[16] Deepak Gupta, “JWT Security Guide: Best Practices & Implementation (2025)”.

[17] Portnox, “What is the Advanced Encryption Standard (AES)?”.

[18] MoldStud, “The Ultimate Checklist for Securing Express.js – Best Practices Explained”.

[19] YittBox, “A Comprehensive Guide to Tailwind CSS: The Utility-First CSS Framework Revolutionizing Web Design”.

[20] GeeksforGeeks, “Tailwind CSS: Utility-First Styling for Rapid UI Development”.

[21] Shivlab, “Tailwind CSS Breakdown: How It Works, Key Advantages, and Top Alternatives”.

[22] Forbes Advisor, “E-Commerce Statistics For India In 2024”.

[23] MDN Web Docs, “Express/Node introduction”.

[24] DEV Community, “Choosing Between Next.js and the MERN Stack: A Simple Guide”.

[25] CodeWalnut, “Pros and Cons of Using Nextjs”.

[26] DEV Community, “Pros and Cons of Next.js”.

[27] GeeksforGeeks, “Middleware in Express”.

[28] Groovy Web, “Express.js vs Next.js: Key Differences”.

[29] Netguru, “Express.js vs Next.js: Key Differences”.

[30] Yaguara, “Global eCommerce Sales Growth From 2021–2027”.

[31] SOAX, “E-commerce growth: How much has e-commerce grown over the years?”.

[32] ResearchAndMarkets.com, “India Ecommerce Business and Investment Databook 2024: Market to Grow by Over $50 Billion During 2023–2028”.

[33] GlobeNewswire, “Growth Trends in India’s E-Commerce Market, 2024–2029”.

[34] AltexSoft, “The Good and the Bad of Express.js Web Framework”.

[35] Node.js Security, “OWASP Node.js Authentication, Authorization and Cryptography Practices”.

[36] DEV Community, “How to Use Axios Interceptors to Handle API Error Responses”.

[37] Razorpay, “Best Payment Gateway in India to Accept Online Payments”.

[38] Wisp CMS, “Should I Host My Next.js Project on Vercel or AWS?”.

[39] ZeroThreat, “Nine Best Practices to Attain Steadfast JWT Security”.

[40] DEV Community, “Leveraging Edge Caching in Next.js with Vercel for Ultra-Low Latency”.

[41] Hashstudioz, “Why Do Some PWAs Feel Slower Than Native Apps? Solving Performance Bottlenecks”.

[42] Mollify LMS, “Performance Optimization — Mollify LMS”.

[43] SearchMyExpert, “PWA Optimization: Performance, Compatibility, Security”.

[44] PageOneFormula, “Using Progressive Web Apps (PWAs): Enhancing User Experience Across Devices”.

[45] Tillitsdone, “Using Axios Interceptors for Error Handling”.

[46] Next.js, “Guides: Authentication”.

[47] Treblle Blog, “How to structure an Express.js REST API – best practices”.

[48] GeeksforGeeks, “REST API CRUD Operations Using ExpressJS”.

[49] Harness, “Introduction to Building a CRUD API with Node.js and Express”.

[50] Teco Tutorials, “Implement Role-based Access Control in a Node.js API”.

[51] International Trade Administration, “eCommerce Sales & Size Forecast”.

[52] tsh.io, “SSR vs SSG in Next.js – tutorial for CTOs and devs”.

[53] AltexSoft, “The Good and Bad of Next.js Full-Stack React Framework”.

[54] Express.js, “Routing – Express.js”.

[55] rowsanali, “How to Build Advanced Ecommerce Product Grids and Detail Pages with Tailwind CSS”.

[56] Strapi, “Top 5 Free Tailwind Component Libraries”.

[57] Nostra AI, “How Website Performance & Page Speed Affects Conversion Rate”.

[58] Noibu, “Breaking down the impact of web performance on ecommerce conversions”.

[59] NitroPack, “How Page Speed Affects Your Conversion Rates”.

[60] Bidnamic US, “How website speed affects your conversion rates”.

[61] Cerbos, “Securing non-human identities: Understanding and addressing the OWASP top 10 threats”.

[62] Node.js Security, “OWASP Node.js Best Practices Guide”.

[63] Alerty, “14 NextJS Security Measures for More Secured Applications”.

[64] OWASP, “API2:2023 Broken Authentication – OWASP API Security Top 10”.

[65] GeeksforGeeks, “JSON Web Token (JWT)”.

[66] GeeksforGeeks, “JWT Authentication In Node.js”.

[67] Fullstack Open, “Fullstack part4 — Token authentication”.

[68] OWASP Belgium, “JWT SECURITY”.

[69] OWASP Cheat Sheet Series, “REST Security – OWASP Cheat Sheet Series”.

[70] Corbado, “Node.js Express JWT Authentication with MySQL & Roles”.

[71] Stack Overflow, “How to choose an AES encryption mode (CBC ECB CTR OCB CFB)?”.

[72] NIST CSRC, “The Galois/Counter Mode of Operation (GCM)”.

[73] Express.js, “Security Best Practices for Express in Production”.

[74] Razorpay, “Razorpay Payment Gateway Flow”.

[75] Razorpay, “Razorpay – Best Payment Solution for Online Payments India”.

[76] Rollout, “Step by Step Guide to Building a Razorpay API Integration in JS”.

[77] Hashnode.dev, “Easy Way to Integrate Razorpay Payment Gateway (Step-by-Step Guide)”.

[78] Razorpay Docs, “Quick Integration – Steps — Razorpay Payment Gateway”.

[79] Sreyas IT Solutions, “Razorpay Webhooks with Node.js”.

[80] Zweck Infotech, “Integrate Razorpay SDK with Express.js / Node.js”.

[81] Vercel, “Vercel Functions”.

[82] Vercel, “Edge Network – Vercel”.

[83] Bird Eats Bug, “How to build and deploy a PWA with Next.js”.

[84] Next.js, “Configuring: Progressive Web Applications (PWA)”.

[85] DEV Community, “Building a Hybrid SQL + NoSQL E-Commerce Data Model”.

[86] Easie, “Database schema design best practices in SQL and NoSQL”.

[87] fabric Inc., “What’s an Example of Good E-Commerce Database Design?”.

[88] InfoQ, “Data Modeling: Sample E-Commerce System with MongoDB”.

[89] Stack Overflow, “MongoDB Schema for ecommerce products that have variations with different SKUs, prices and stock”.

[90] Express.js, “Express 5.x – API Reference”.

[91] RisingStack Engineering, “10 Best Practices for Writing Node.js REST APIs”.

[92] ProfileTree, “A Guide to Mobile-first Website Design for Lead Generation”.

[93] GeeksforGeeks, “How To Implement JWT Authentication in Express App?”.

[94] StackHawk, “NodeJS Broken Authentication Guide: Examples and Prevention”.

[95] GeeksforGeeks, “Razorpay Payment Integration using Node.js”.

[96] Razorpay Docs, “Integration Steps — Node.js SDK — Razorpay Docs”.