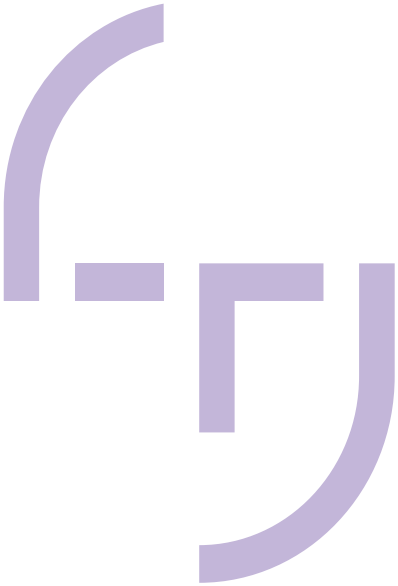
Tampere University of Applied Sciences



Design and Development of an Online Shopping Platform for Badminton Equipment

Kiet Nguyen

Long Le

BACHELOR'S THESIS
May 2025

Degree Programme in Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Bachelor's Degree Programme in Software Engineering

LE, LONG & NGUYEN, KIET:
Design and Development of an Online Shopping Platform for Badminton Equipment

Bachelor’s thesis, 48 pages, appendices 9 pages
May 2025

The thesis focuses on leveraging contemporary web technologies to build and develop an online retail website. Web server deployment, database administration, secure payment systems, responsive design, and front-end and back-end development are important domains. The goals were to design a professional, efficient platform, add features like shopping carts, payment gateways, user authentication, and product listings, test it for performance, security, and usability, and record the development process for academic and practical usage. The results include a fully functional website, source code, development process documentation, and a final thesis paper outlining the effort, goals, and lessons learned.

The theoretical part of the thesis focuses on the technologies used for the project, providing a comprehensive overview of the foundational principles, architectural patterns, and modern frameworks that play important roles in the development of full-stack web applications. This includes an in-depth analysis of the client-server model, RESTful API design, and the interaction between the frontend and backend.

Key words: retail, website, secure

CONTENTS

ABBREVIATIONS AND TERMS

API	Application Programming Interface
AI	Artificial Intelligence
BSON	Binary JavaScript Object Notation
B2B	Business-to-business
B2C	Business-to-consumer
BNPL	Buy now, pay later
CSS	Cascading Style Sheets
DOM	Document Object Model
ECMA	European Computer Manufacturers Association
HTTP	HyperText Transfer Protocol
HTML	HyperText Markup Language
IANA	Internet Assigned Numbers Authority
I/O	Input/Output
JS	JavaScript
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
JWA	JSON Web Algorithm
NoSQL	Not only SQL
NPM	Node Package Manager
SQL	Structured Query Language

1 INTRODUCTION

1.1 Motivation and background about the project

The rapid development of technology over the past few decades has changed the way society approaches business. What was once a futuristic vision of an interconnected digital world has now become an indispensable part of everyday life. At the heart of the rapidly changing world lies the Internet, the most dynamic and expansive creation for information exchange. Today, nearly every aspect of human activity, from education and electronic commerce (e-commerce) to entertainment and governance, has been significantly influenced by the Internet (OECD, 2019).

In the rapidly changing world of technology, web applications have also become an essential part of daily life. These applications have evolved from simple static web pages to more sophisticated, interactive platforms, capable of delivering a tailored user experience. The journey of web applications began in 1995 with the introduction of JavaScript (JS) by Brendan Eich. JavaScript was first developed for Netscape 2 and became the ECMA-262 standard in 1997. This is a major milestone that later enables dynamic content and interactivity on the web. Modern web applications are now used for a wide variety of purposes, such as e-commerce and online communication (W3schools, n.d).

1.2 Overviews and objectives of the project.

The primary aim of this thesis project is to design, develop, and evaluate a fully functional **e-commerce web application** utilizing a modern, full-stack JavaScript technology stack. The system is intended to build an e-commerce store, providing a comprehensive set of features including product browsing, shopping cart management, user authentication, secure payment processing, and responsive design for both desktop and mobile platforms. The project aims to contribute to the e-commerce industry by providing a safe and fast platform for online shopping.

The project integrates **React.js** for client-side interface development, **Tailwind CSS** for responsive and utility-first styling, **Node.js and Express** for backend logic, **MongoDB** as a NoSQL database solution, and **Stripe API**, an Application Programming Interface for payment gateway integration. The architectural design follows a modular, scalable pattern to ensure maintainability and extensibility.
This implementation also aims to reflect professional software development practices by incorporating agile project management methodologies, version control through Git, and continuous testing and evaluation throughout the software development lifecycle.

The main objective of the project is to develop a web application that can provide a safe online shopping experience for users, focusing on these criteria:

- Ensuring safe online transactions through robust authentication and encrypted payment systems to protect user information.
- Providing a responsive, intuitive interface that allows users to easily browse, select, and purchase products across devices.
- Providing convenient and time-saving access to a wide variety of goods through a centralized digital platform.

2 THEORETICAL BACKGROUND AND TECHNOLOGIES USED

2.1 E-Commerce Development Principles

E-commerce is the purchase and sale of goods and services, or the exchange of money or information via electronic networks. These e-commerce transactions usually consist of four categories: Business-to-business (B2B), Business-to-consumer (B2C), consumer-to-consumer, and consumer-to-business. The term e-commerce is sometimes called e-business. The transactional procedures that comprise online retail shopping are frequently referred to as e-tail (Hashemi-Pour, 2023).

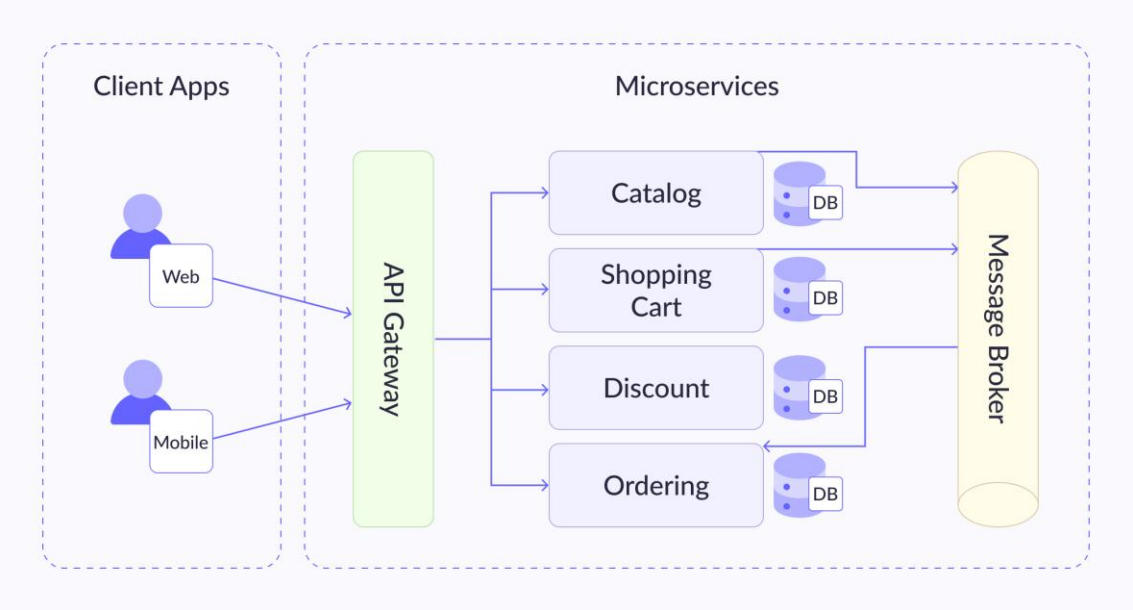


Figure 1. Microservice Architecture of an E-commerce (Demkovych, 2023).

The primary purpose of e-commerce web design is to offer a visually appealing and intuitive platform for both purchasing and selling goods, as well as providing a customized customer experience. Customers can go through a wide variety of products in the same place without having to worry about time and other aspects of an offline store. This is a very crucial aspect that must be considered when designing an e-commerce website to retain customers and welcome new ones. Having a well-designed page for ordering, navigation, and looking at a variety of

products is essential to give a competitive business advantage (GeeksforGeeks, 2024).

Designing an effective e-commerce website hinges on principles such as minimalistic design to reduce distractions, ease of use, clear and consistent branding to reinforce identity, responsiveness across devices ensures accessibility for all users, and product representation that reflects accurate quality and builds user trust. When properly applied, these concepts offer several advantages. For instance, companies may reach clients in large geographic areas, providing seamless and consistent user experiences. Additionally, businesses gain from lower operating expenses as a result of automation and digital infrastructure, while customers save time pursuing and making purchases, enjoy a large selection of products, and have various payment options. Nevertheless, e-commerce companies still have to deal with issues like growing competitiveness, the difficulty of integrating Artificial Intelligence (AI), unstable supply chain, cybersecurity risks, and changing customer demands. Resolving all these challenges is essential to maintaining competitiveness in the rapidly changing digital market (Tully, 2025).

2.2 Introduction to MERN stack

The project utilizes the MERN stack to implement the online store, a popular JavaScript-based technology, which consists of MongoDB, Express, React.js, and Node.js with the addition of Tailwind CSS and TypeScript for type safety. MongoDB, a NoSQL database, stores user and activity data in a flexible, document-orinated format that supports scalability and fast access. Express is a lightweight framework running on Node.js, used to build server-side logic and manage routing between the client, server, and database. While Node.js provides the runtime environment for executing JavaScript code on the server side, enabling seamless integration between front and back ends. React.js powers the dynamic, single-page user interface through usable components, creating a responsive and interactive user experience. Tailwind CSS, a Cascading Style Sheets (CSS) framework, enhances the UI development by offering ultility-first classes for raped and consistent styling. In addition, TypeScript introduces static

typing to improve code quality and maintainability. Figure 2 illustrates the functional architecture of the MERN stack through a diagrammatic representation. Together, these technologies enable the creation of a high-performance, scalable, and user-friendly e-commerce platform (Swain, 2025).

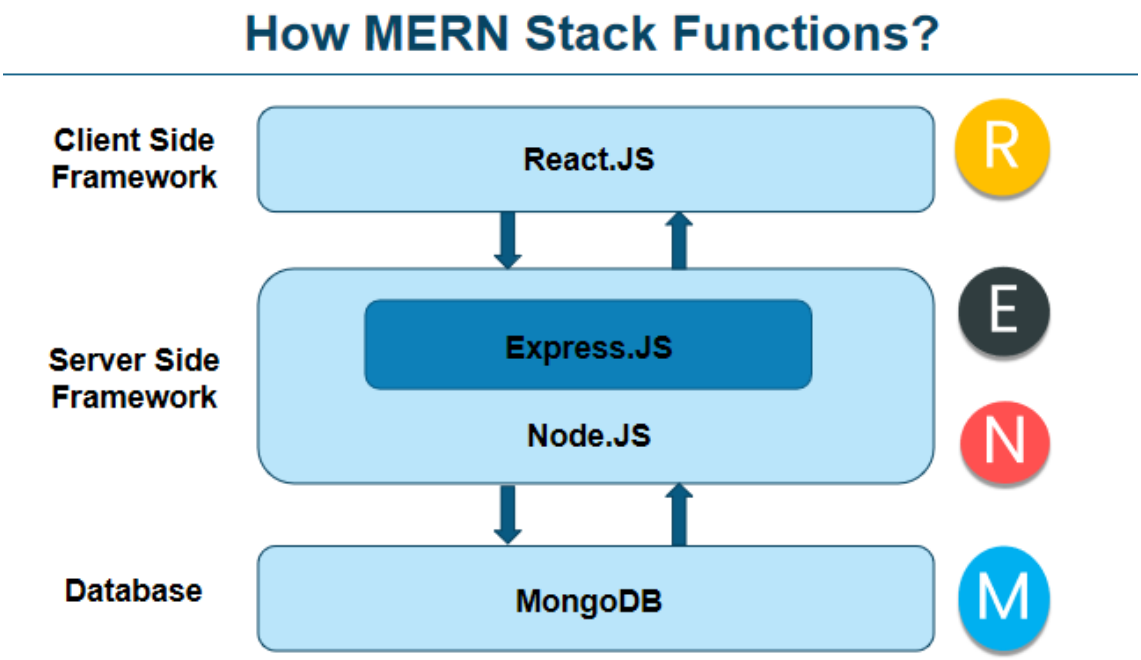


Figure 1. How the MERN stack works together (Swain, 2025).

2.3 Technologies used for Front End and Back End

2.3.1 React.js

For this project, React, a JavaScript library maintained by **Meta** (formerly Facebook), was selected for its component-based architecture, efficient rendering via the virtual DOM, and strong community support. React's component-based structure allows developers to build encapsulated components that manage their scalability and maintainability across the application. The virtual DOM optimizes rendering by updating only the components that have changed, leading to improved performance. React's declarative syntax simplifies the development and debugging of complex UI states, making the code more predictable and easier to understand. Figure 3 provides an example of React code to illustrate the declarative syntax in practice. Additionally, React offers

several advantages, including backward compatibility, performance optimization, SEO-friendliness, and flexibility. These features collectively contribute to React's effectiveness in modern frontend development. (Geeksforgeeks, 2025)

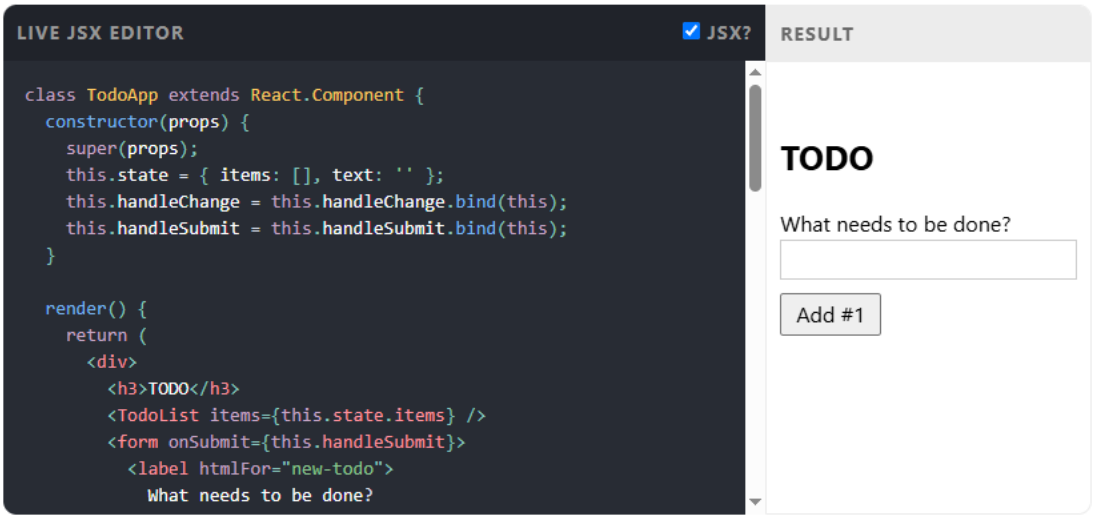


Figure 3. Example of React.js code (Reactjs.org, n.d, edited).

2.3.2 Tailwind CSS

To complement React, **Tailwind CSS** is employed for its utility-first approach to styling. Unlike traditional CSS frameworks, Tailwind allows developers to apply design elements directly within the HyperText Markup Language (HTML) or JavaScript XML (JSX), reducing the need for custom stylesheets and speeding up the design workflow. Figure 4 presents a code comparison between Tailwind CSS and traditional CSS to highlight their structural differences and styling approaches. Tailwind ensures responsive design, allowing the application to adapt fluidly to various screen sizes and devices, thereby improving user accessibility and experience. Together, React and Tailwind CSS provide a robust and efficient framework for building a visually appealing, responsive, and user-friendly frontend interface (Roy, 2022).

Figure 4. Differences between Tailwind CSS and Normal CSS (Valerie, 2024).

2.3.3 Node.js

Node.js is an open-source and cross-platform JavaScript runtime environment that runs the V8 JavaScript engine – originally developed for Google Chrome – outside of the web browser. This allows Node.js to run at high performance. Unlike traditional server environments, Node.js operates within a single-threaded event loop. It consists of non-blocking, asynchronous input/output (I/O) operations that prevent the execution thread from becoming idle during input/output tasks. Node.js registers callbacks and resumes execution only when responses are received, allowing the system to efficiently manage numerous concurrent connections. Furthermore, it supports modern ECMAScript features without dependency on browser updates, giving developers control over language features through Node.js version and runtime flags (Nodejs.org, n.d.).

As a backend technology, Node.js offers some advantages such as high performance, scalability, and a lightweight structure that supports microservices. Node.js architecture is illustrated in Figure 5, highlighting its event-driven model and interaction flow. Node.js is well-suited for applications with intensive I/O

needs and contributes significantly to overall web application performance, reliability, and maintainability (Rachowicz, 2024).

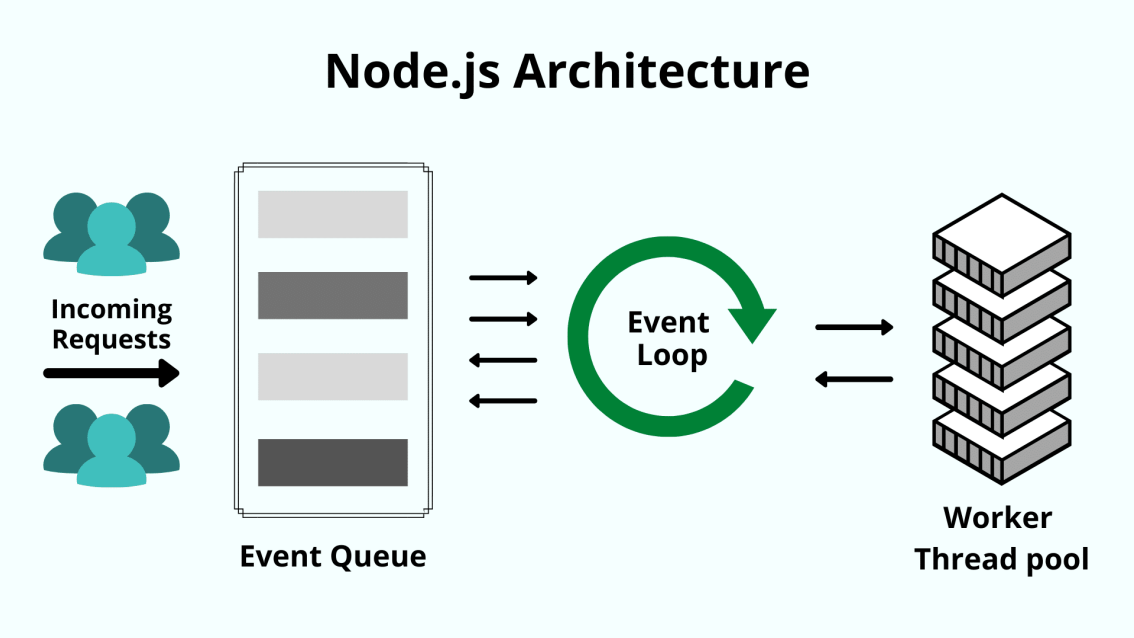


Figure 5. Illustration of a Node.js Architecture (Kinsta, 2023).

2.3.4 Express

Express is a Node.js web application framework that provides features to build a Backend for a project. It is used in single-page, multipage, and hybrid web applications. It also plays an important role in managing servers and routing. Express.js was chosen for the project because of its ease of implementation. The main features of Express are fast server-side development, middleware, routing, and asynchronous. As illustrated in Figure 6, providing information on how Express works. Express is written in JavaScript and is easy to learn. Figure 7 presents a sample of Express code, while Figure 8 displays the corresponding output generated by this code (Sharma, 2024).

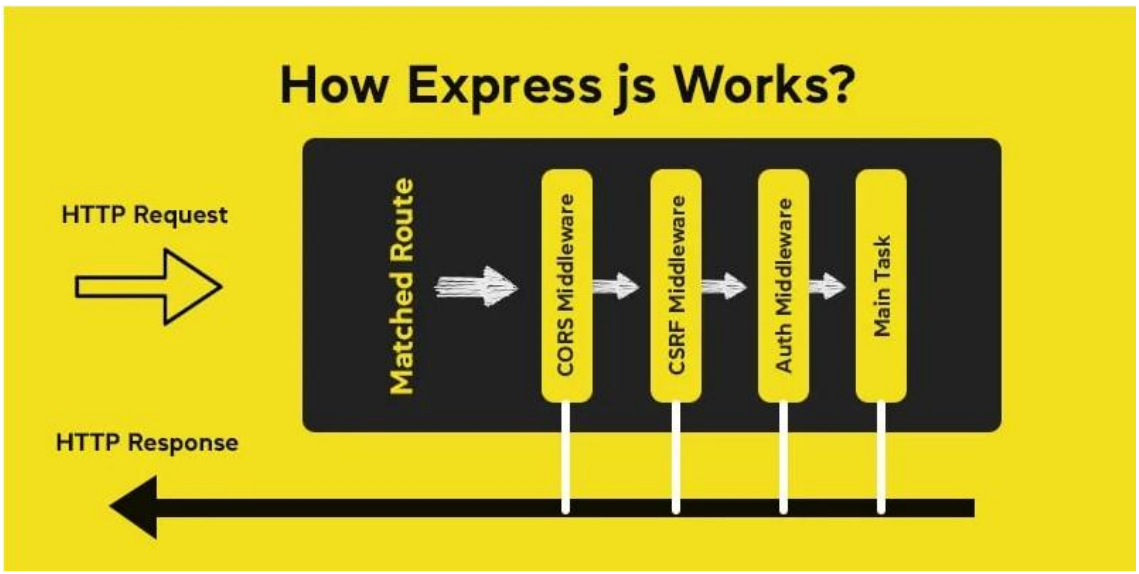


Figure 6. How Express works (Panchal, 2024).

```
const express = require('express');
const app =express();

app.get('/',(req,res) => {
    res.send("Welcome to simplilearn");
});

app.listen(4000, ()=>{
    console.log("listening to port 4000");
});
```

Figure 7. Express framework example code (Sharma, 2024).

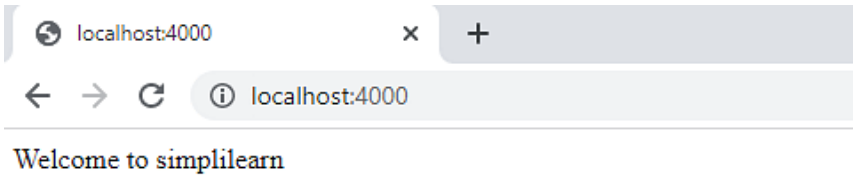


Figure 8. Express output from example code (Sharma, 2024).

2.3.5 TypeScript

TypeScript, a robust programming language, was developed by Anders Hejlsberg of Microsoft. Since its release, the language has become a major tool that boosts the JavaScript development experience. TypeScript requires more strict typing, but reduces runtime errors. Its unique forcing type annotations ensure any data object that interacts with the interface must have a consistent design. The strict typing is demonstrated in Figure 9 when compared to JavaScript. TypeScript can be simplified as JavaScript with strict types. Sharing the similarity makes the TypeScript language integrate with modern frameworks' technologies that use JavaScript, such as Express, React, and Vue.js. The TypeScript rigid structure was essential to keep the code flow, and was chosen and used for this project. (Tomar, 2025).

Figure 9. TypeScript and JavaScript side-by-side comparison (Typescriptlang.org, n.d).

2.3.6 MongoDB

MongoDB was chosen as the database management system for the project. Due to being a NoSQL database, MongoDB has a Schema-less design, which means that the data table is more flexible and very suitable for documents data have multiple structures or fields. MongoDB stores data in a collection of Binary JavaScript Object Notation (BSON), a human-readable, text-based format that is straightforward to use. Figure 10 illustrates a MongoDB architecture and how the

system interacts with each other. In addition, BSON modern data handling can optimize the storage data speed. For all its advantages in managing a variety of dynamic data for projects, MongoDB has played a vital role as the database option in building fast and flexible applications (Zhang, 2025).

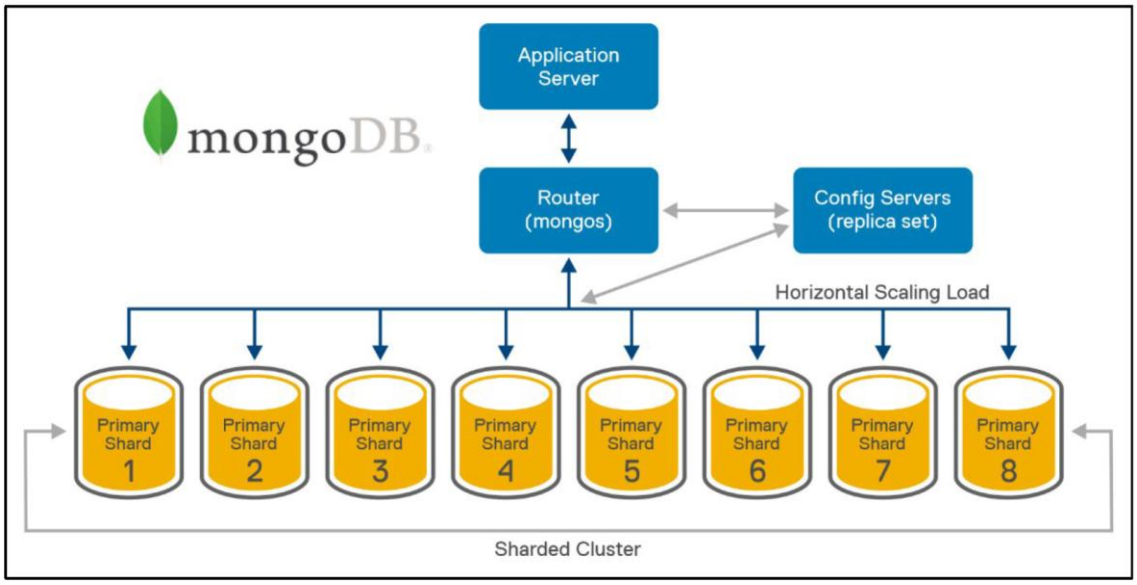


Figure 10. MongoDB architecture (DellTechnologies, n.d.).

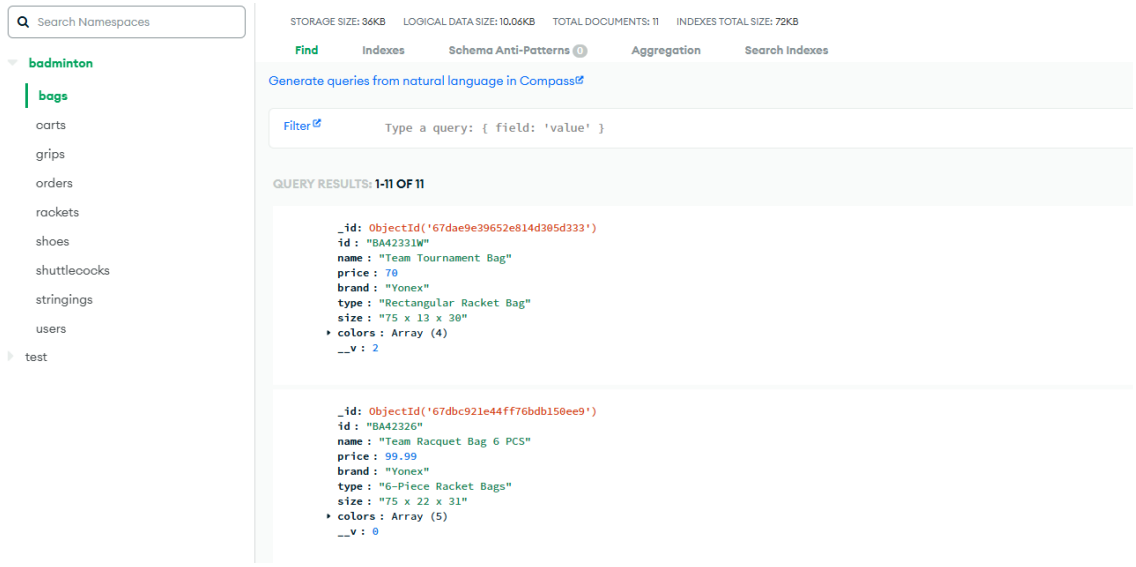


Figure 11. MongoDB collections example.

```
const { MongoClient, ServerApiVersion } = require('mongodb');
const uri = "mongodb+srv://kietnguyen:<db_password>@shoppingbadminton.44m9i.mongodb.n

// Create a MongoClient with a MongoClientOptions object to set the Stable API versio
const client = new MongoClient(uri, {
  serverApi: {
    version: ServerApiVersion.v1,
    strict: true,
    deprecationErrors: true,
  }
});

async function run() {
  try {
    // Connect the client to the server (optional starting in v4.7)
    await client.connect();
    // Send a ping to confirm a successful connection
    await client.db("admin").command({ ping: 1 });
    console.log("Pinged your deployment. You successfully connected to MongoDB!");
  } finally {
    // Ensures that the client will close when you finish/error
    await client.close();
  }
}
run().catch(console.dir);
```

Figure 12. Sample code for connection.

While Figure 11 illustrates a MongoDB collection example, Figure 12 provides a sample code for the connection to the Database can be established seamlessly, allowing developers to interact with data efficiently. This facilitates operations such as querying, updating, and managing data in a structured and reliable manner. Tools such as Mongoose for MongoDB allow for schema validation, middleware support, and simplified CRUD operations while maintaining consistent and type-safe integration with the application backend.

2.3.7 NoSQL

MongoDB is the pioneer of the databases now known as NoSQL databases, which include document-oriented databases like MongoDB. It features flexible schemas that do not require determining a fixed schema for the data, making it ideal for semi-structured and unstructured data. NoSQL schema also scales horizontally and uses range or hash distributions. This feature of NoSQL schema

helps with scalability and performance in the long run (Cloud.google, n.d). Figure 13 provides us key features comparison between traditional SQL and NoSQL.

Feature	SQL and NoSQL database difference	
	SQL	NoSQL
Data model	Structured	Unstructured or semi-structured
Scalability	Vertical scaling	Horizontal scaling
Performance	Complex queries can be slow	Fast read and write performance
ACID compliance	Yes	No or partial
Schema	Rigid	Flexible
Language	SQL (Structured Query Language)	JSON (JavaScript Object Notation), XML, YAML, or binary schema

Figure 13. SQL and NoSQL differences (Chernetskiy, 2024).

2.3.8 NPM

NPM, which stands for Node Package Manager, is an essential tool for every Node.js project. It serves as a central repository for JS packages, which are pre-written code modules that can be imported into a project, saving time and effort. This ensures project compatibility as well as keeping it updated with the latest tools and technologies. NPM is used to help with dependency management, which can smooth the JavaScript development while also providing version control for the packages. NPM utilizes CLI to initialize a project and install and uninstall packages (Singh, 2024).

2.4 Security and Payment Integration

2.4.1 JWT

JWT, defined as JavaScript Object Notation (JSON) Web Token, is an open standard for encrypting data that is transferred between parties. It ensures data

integrity as well as its authenticity. JWTs are widely used in API authentication and authorization workflows. The three components that make up the JWT structure are: a header, a payload, and a signature (Bello, 2023). The header contains information about the overall JWT, such as the algorithm used (alg) and the media type of JWT (typ), and sometimes the content type (cty). As seen in Figure 14, each part consists of a Base64-encoded string with periods between characters.

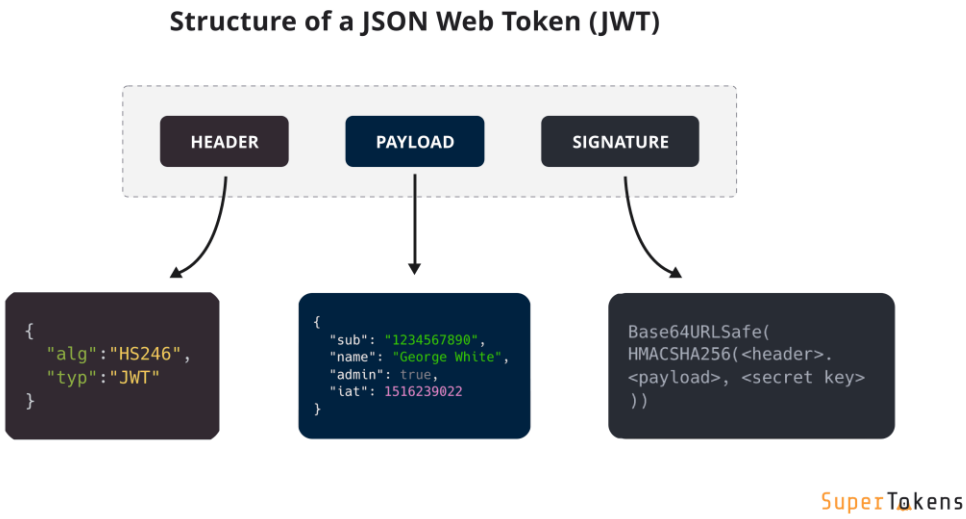


Figure 14. Structure of a JWT (Ibrahim, 2024).

2.4.2 Bcrypt

Bcrypt is a hash function based on the Blowfish cipher, widely used for safely storing passwords in backend systems. It performs a one-way transformation, converting passwords into fixed-length hash values that cannot be reversed to their original form, thereby reducing the risk of unauthorized access. Each time a user attempts to log in, bcrypt hashes the input password and compares the result with the stored hash to validate the credentials. Additionally, bcrypt incorporates a randomly generated value known as a “salt” into the hashing process, ensuring that even identical passwords produce unique hashes (Arias, 2021).

The Bcrypt hashing process relies on three key input parameters: the password string, a salt value, and a cost factor. The password string, limited to a maximum

of 72 bytes, is the user-provided input that undergoes hashing. The salt is a randomly generated 16-byte value that is prepended to the password before hashing, enhancing uniqueness and protecting against precomputed attacks such as rainbow tables. The cost factor, also known as the work factor, is a numerical value that determines the computational complexity of the hashing process by specifying how many iterations the algorithm performs. A higher cost factor increases security by making brute-force attacks more time-consuming. Together, these inputs ensure that each hash output is both secure and unique (Buchanan, 2025). Figure 15 illustrates an example of bcrypt in practice.

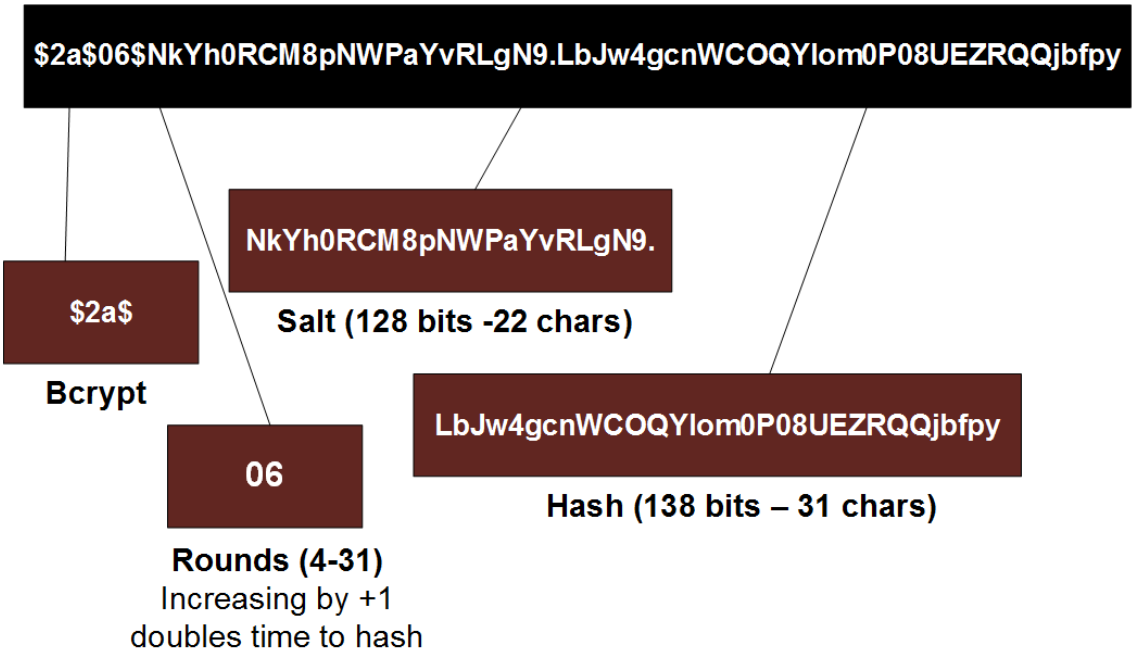


Figure 15. Example of Bcrypt hashing (Buchanan, 2025).

2.4.3 Stripe

Stripe is an online payment processing and credit card processing system for businesses. It allows online payment through credit and debit cards. Stripe will process customers' card information and provide it to the acquirer – a bank that processes the sale on the business's behalf. After the issuing bank finalizes its approval, the funds will then be transferred from the Stripe account to the business bank account, with some additional fees. As of right now, Stripe

supports credit cards, debit cards, bank transfers, Apple Pay, Google Pay, and BNPL (Payn, 2025). Key features of Stripe are presented in Figure 16.

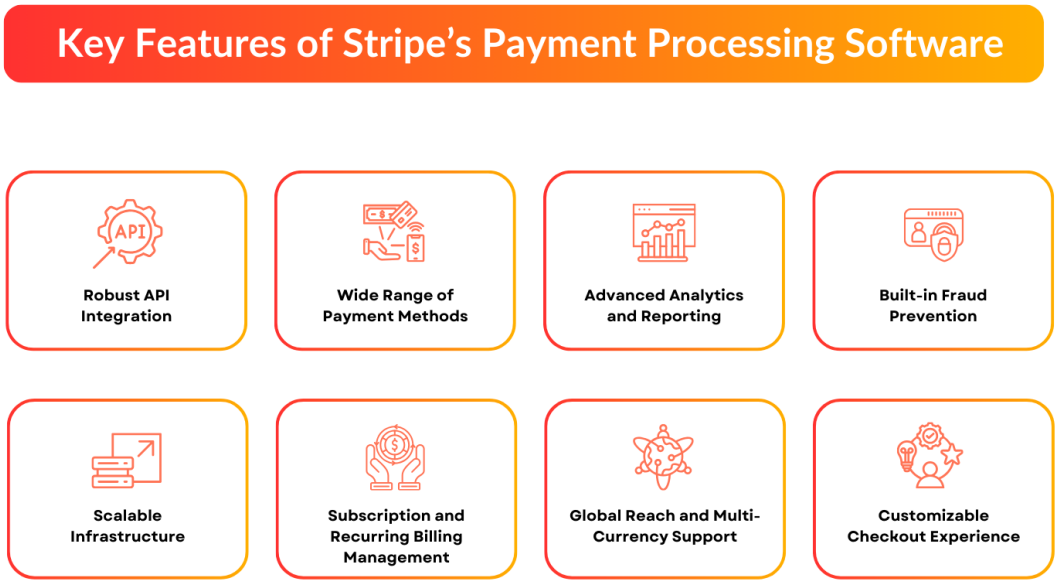


Figure 16. Stripe key features (InfoStride, 2024).

As e-commerce platforms continue to grow, ensuring security measures is essential to protect user data and uphold consumer trust. Foundational principles such as privacy, integrity, authentication, and non-repudiation are vital to safeguarding transactions and preventing malicious activities. This project integrates Stripe as the primary payment processing service, due to its comprehensive suite of security features and developer-friendly architecture. Stripe supports a wide range of payment methods, including credit cards, mobile wallets, and international options, making it ideal for scalable and globally accessible platforms. Its instantaneous transaction processing, webhook support, and detailed reporting tools enhance operational efficiency and user experience. Furthermore, Stripe employs advanced fraud detection algorithms, PCI DSS compliance, and tokenization techniques, significantly reducing the risk of financial breaches. The ease of integration, extensive documentation, and strong community support further solidify Stripe as a reliable, secure, and future-proof solution for handling online payments in modern web applications (Varghese, 2025).

3 DEVELOPMENT PROCESS

3.1 Initial Planning and Setup

The thesis project was planned as a collaboration between Kiet Nguyen and Long Le, aiming to develop a scalable and functional online retail website using modern web technologies. The project focused on core e-commerce features such as product listings, shopping carts, user authentication, and secure payment systems, with full documentation intended for academic and practical use. Planning included selecting tools like React and Tailwind CSS for the frontend, Node.js and MongoDB for the backend, and Git with GitHub for version control. The team adopted an Agile approach with weekly milestones, component-based architecture, RESTful APIs, and responsive design. Long Le was responsible for planning all the ideas to code, while Kiet Nguyen was responsible for the initial setup, including creating the GitHub repository, structuring the project folders, and designing the database schema. These early steps ensured a consistent development process aligned with the project's goals, which include delivering a fully functional website, source code, and a final thesis report useful to future developers and small business owners.

3.2 Task Distribution and Collaboration

After the initial setup, both team members moved on to the development of the application. The core functionalities of the frontend were handled by Long Le, including implementing the user interface for the application. Kiet Nguyen was responsible for implementing the backend code, which included setting up the backend server using Node.js and Express, creating REST API endpoints, setting up user authentication, as well as handling the database structure and seeding the data. Later on in the project, Long Le helped with implementing the Stripe API for the payment integration with further improving the authentication logic.

Once the core backend features were implemented, Kiet Nguyen transitioned to focusing on the thesis report, while Long Le continued with the development of the application. During this phase, Long Le was responsible for implementing and

documenting various technical aspects of the project, including system design, architectural decisions, while Kiet Nguyen was accountable for project goals, testing procedures, system diagrams, and the final evaluation. This ensured that both the practical development and theoretical components of the thesis were thoroughly addressed.

This division of roles allowed each team member to work efficiently and focus on their main tasks while still maintaining good communication as well as coherence between the software and the report. Discord was used for day-to-day communication and quick discussions, as well as Microsoft Teams for saving documents. In addition, face-to-face meetings were held semantically for debugging, brainstorming, reviewing the process, planning upcoming tasks, and creating new features. GitHub was used for version control, ensuring transparency and synchronization between members.

Although the division of tasks remained consistent throughout the project, there was support between team members with their respective tasks. Kiet Nguyen contributed to debugging and implementing various Front End and Back End features, while Long Le also helped with reviewing and giving feedback on the report. This flexibility helped tremendously to address some challenges during the development process. Each member was still able to focus on their main responsibilities while still maintaining communication as well as shared accountability.

4 IMPLEMENTATION DETAILS

4.1 System Architecture

The project consists of a high-level functional workflow of a badminton shop, which is structured around three primary actors: **customer**, **administrator (seller)**, and the **application system** itself. The flow is divided into multiple stages, reflecting authentication, product browsing, cart management, order processing, and administrative functions.

4.1.1 User Interaction and Authentication

The system initiates with a central start node, from which both customers and administrators diverge based on their roles:

- **Customers** begin by browsing available products. Upon selecting an item, product details and pricing are displayed. To proceed with transactions, the customer must authenticate through a sign-up or login process using an email address.
- **Administrators (Sellers)** authenticate using a secure Administrator Token. Upon successful authentication, they are granted access to product management functionalities, including viewing and modifying the product listings.

4.1.2 Customer Activities After Authentication

Once authenticated, customers gain access to enhanced functionalities. Authenticated customers can add products to a personal shopping cart, with each item persistently stored and directly associated with the individual customer's profile. The shopping cart contents can be reviewed at any time through the user interface, allowing for informed purchasing decisions. Initiating the checkout process triggers the creation of a new order in the backend system, making the formal transition from product selection to the commencement of order processing.

4.1.3 Order Processing and Fulfillment

The **Order Processing** section encapsulates both customer-facing and back-end operations:

- Orders are created and stored, including customer information and selected items.
- The administrator interface allows sellers to view customer orders, prepare shipments, and update order statuses through various stages: “Processing,” “Shipped,” and finally “Delivered”.
- Each transition in product status is reflected in the order history accessible to both the customer and the administrator.
- Once the product is delivered, payment is processed for cash-on-delivery (COD) transactions, and the order lifecycle concludes with inventory updates and order closure by the administrator.

4.1.4 System Synchronization and Data Flow

The central application manages interactions across user roles by interfacing with the **product database**, **cart modules**, and **order tracking system**. Data synchronization ensures that product availability, cart contents, and order statuses are accurately reflected in up-to-date across all interfaces. Figure 17 outlines the interaction between the customers, the badminton shopping application, and the administrators (sellers). The flowchart illustrates the key processes involved in product browsing, authentication, cart management, order processing, and product delivery.

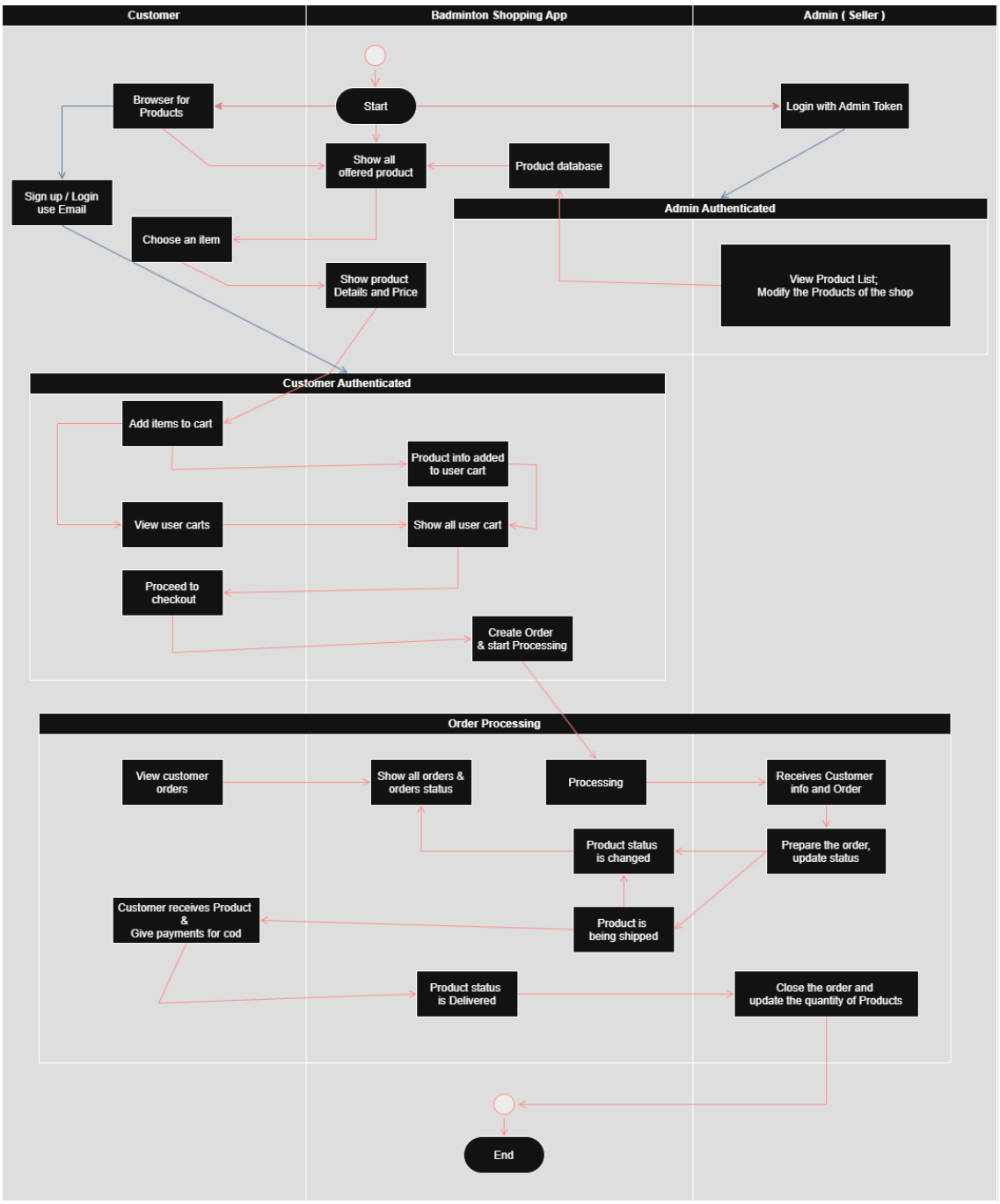


Figure 17. Customer and Administrator Workflow in Badminton Shopping App

4.2 Key Features and Functionalities

This section outlines the core features and functionalities of the application. The primary objective of this project is to deliver a secure, user-friendly, and efficient e-commerce experience for both customers and business owners. Each functionality was built with modularity and scalability to ensure better maintainability.

4.2.1 Authentication and Security

User authentication and role-based access control have been implemented to support secure and functional interactions between customers and owners. The backend utilizes JWTs to manage user sessions securely. To prevent unauthorized access, all sensitive routes are guarded using middleware that verifies the authenticity of JWT tokens and checks for the user's roles. This mechanism ensures that only authenticated users can perform role-specific actions such as managing stocks and orders.

```
// Function to extract token from the Authorization header
const getTokenFrom = (request: Request): string | null => {
    const authorization = request.header('Authorization');
    if (authorization && authorization.startsWith('Bearer ')) {
        return authorization.replace('Bearer ', '');
    }
    return null;
};

export const authenticateUser = (
    req: AuthRequest,
    res: Response,
    next: NextFunction
): Response | void => {
    const token = getTokenFrom(req);
    if (!token) {
        return res.status(401).json({ error: 'Access denied' });
    }

    try {
        const verified = jwt.verify(token, config.JWT_SECRET) as
jwt.JwtPayload;
        req.user = verified;
        next();
    } catch (error) {
        return res.status(400).json({ error: 'Invalid token' });
    }
```

```
};

// Middleware to check if the user is an administrator
export const isAdmin = (
    req: AuthRequest,
    res: Response,
    next: NextFunction
): Response | void => {
    if (req.user && req.user.role === 'admin') {
        next();
    } else {
        return res.status(403).json({ error: 'Access denied.
Administrators only.' });
    }
};
```

Code snippet 1. Authentication middleware code.

The Code snippet 1 demonstrates the implementation of authentication and authorization logic using the jsonwebtoken (JWT) library in a Node.js backend. The middleware first extracts the token from the Authorization header of incoming HTTP requests. It then verifies the token using a secret key defined in the configuration. If the token is missing or invalid, the request is rejected with an appropriate HTTP error code. In addition, a separate middleware function checks the authenticated user's role to determine if they have administrator privileges. Only users with the role of "administrator" are granted access to protected routes for managing product inventory and processing orders. This approach ensures secure access control based on verified user credentials and roles.

```
const userSchema = new Schema<User>({
    email: {
        type: String,
        required: true,
        unique: true,
        match: /.+\@.+\..+/,
    },
    password: {
        type: String,
```

```
        required: true,
    },
    role: {
        type: String,
        enum: ['admin', 'customer'] as UserRole[],
        default: 'customer',
    },
    createdAt: {
        type: Date,
        default: Date.now,
    },
}, { collection: 'users' });
```

Code snippet 2. User schema code model.

Code snippet 2 shows the user model being implemented. It consists of the user's email address, password, created time, and the role to which the users are assigned. New users will be defaulted to being a "customer" as shown in the model.

```
// User routes
router.post('/', authenticateUser, createOrder);
router.get('/user', authenticateUser, getUserOrders);
router.get('/:id', authenticateUser, getOrderById);
router.post('/:id/cancel', authenticateUser, cancelOrder);

// Admin routes
router.get('/admin/all', authenticateUser, isAdmin, getAllOrders);
router.put('/admin/:id/status', authenticateUser, isAdmin,
updateOrderStatus);
router.put('/admin/:id/payment', authenticateUser, isAdmin
updatePaymentStatus);
router.delete('/admin/:id', authenticateUser, isAdmin, deleteOrder);
```

Code snippet 3. Application routes.

The website's authentication logic is demonstrated in Code snippet 3. Some routes will require administrator rights to access. The system will first check for the user's role. If the user is a customer, they will only see the main website. If

the user has an administrator role, they are granted access to role-specific routes, including order management, payment status tracking for individual orders, and stock quantity management.

4.2.2 Secure Payment and Delivery

The payment process is handled securely with the Stripe API, allowing customers to make secure online payments. Each transaction is processed with order details recorded in the database for both the customer and administrator references. Once the payment is confirmed, an order is generated, and the delivery process begins. Customers can also track the present status of their payment and order.

```
export const createPaymentIntent = async (req: Request, res: Response) =>
{
    try {
        const { orderId } = req.body;

        // Get order details
        const order = await OrderModel.findById(orderId);
        if (!order) {
            return res.status(404).json({ success: false, message: 'Order
not found' });
        }

        // Create payment intent
        const paymentIntent = await stripe.paymentIntents.create({
            amount: Math.round(order.totalAmount * 100), // Convert to
cents
            currency: 'usd',
            metadata: {
                orderId: order._id.toString(),
                orderNumber: order.orderNumber
            }
        });

        // Update order with payment intent ID
        order.paymentIntentId = paymentIntent.id;
```

```
        await order.save();

        res.json({
            success: true,
            clientSecret: paymentIntent.client_secret
        });
    } catch (error) {
        console.error('Error creating payment intent:', error);
        res.status(500).json({ success: false, message: 'Error creating payment intent' });
    }
};
```

Code snippet 4. Stripe payment code.

Code snippet 4 demonstrates the payment processing function. The system will first process the details and status of the order. Figure 18 displays the customer's order checkout interface. A payment instance is then initiated, such in Figure 19, and upon successful payment, the order status is updated with the corresponding payment ID. Subsequently, the shipping process starts.

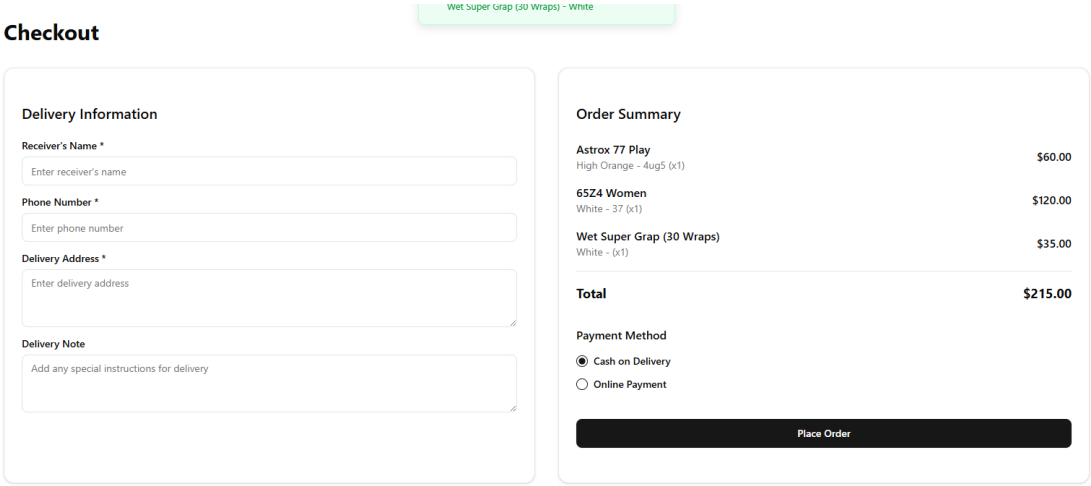


Figure 18. Checkout interface.

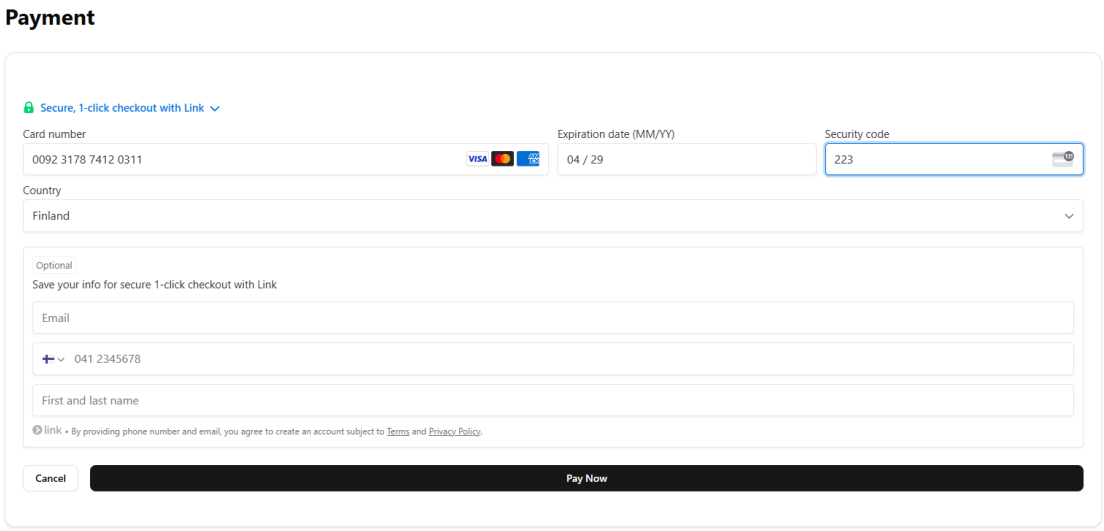


Figure 19. Payment interface

4.2.3 Separate Customer and Administrator Interfaces

The application currently offers two role-specific interfaces: Customer UI and Administrator UI. Figure 20 provides the home page interface that users meet when first opening the website. On the top left of the screen, there are 7 different tabs, each representing a type of product that is being sold. The user can navigate to different tabs to view the available products. Figure 21 displays the different types and models of badminton rackets.

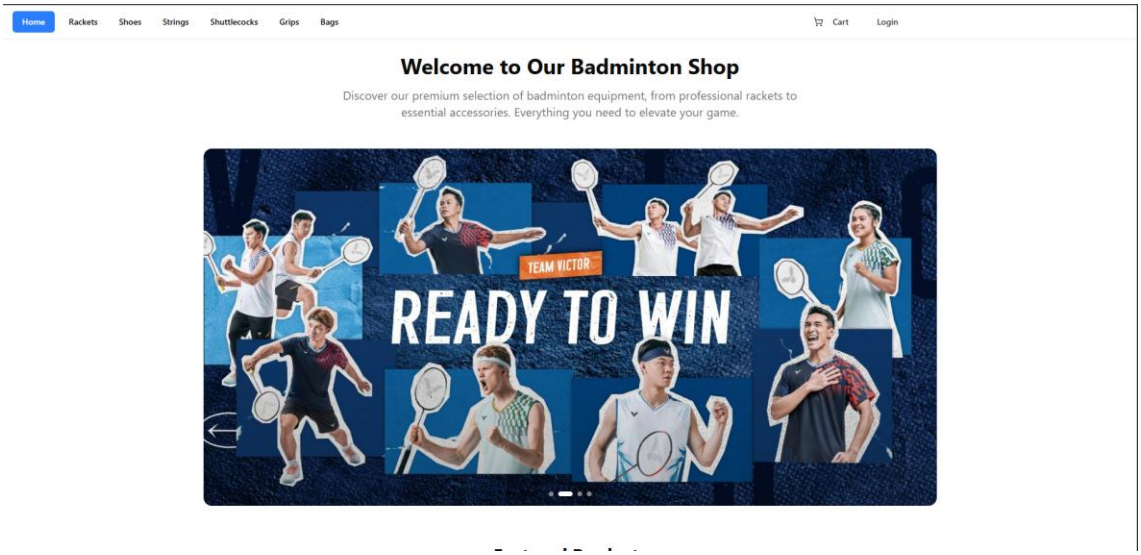


Figure 20. Application home page.

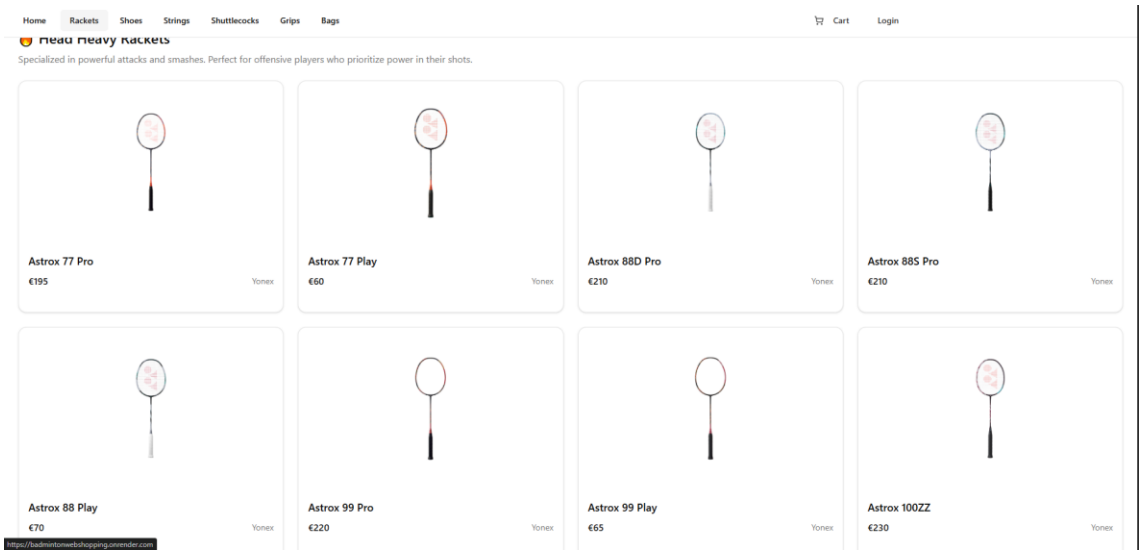


Figure 21. Rackets product page.

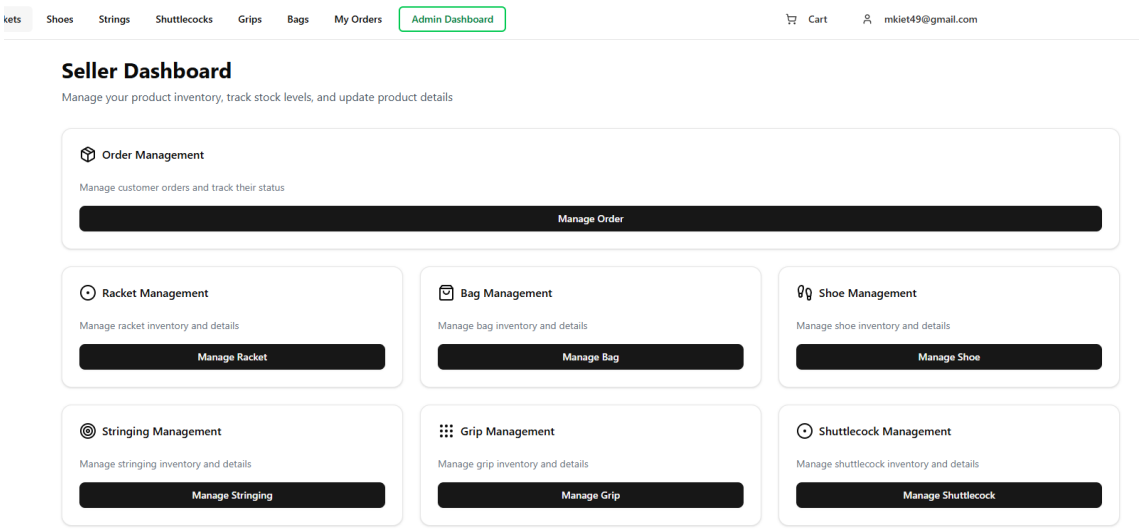


Figure 22. Administrator Dashboard.

For administrators, there will be a different dashboard as displayed in Figure 22. In this administrator interface, there are options for managing product stock and for managing customers' orders. Figure 23 demonstrates a full order, which contains the customer's information, the payment information, and the details of their order. The administrator can change the status of the order and the payment status, ensuring instant communication between the customer and the administrator. In addition, Figure 24 displays the data table of Racket to support the administrator managing the shop data without the necessary to go to different applications or a database manager to see the data info.

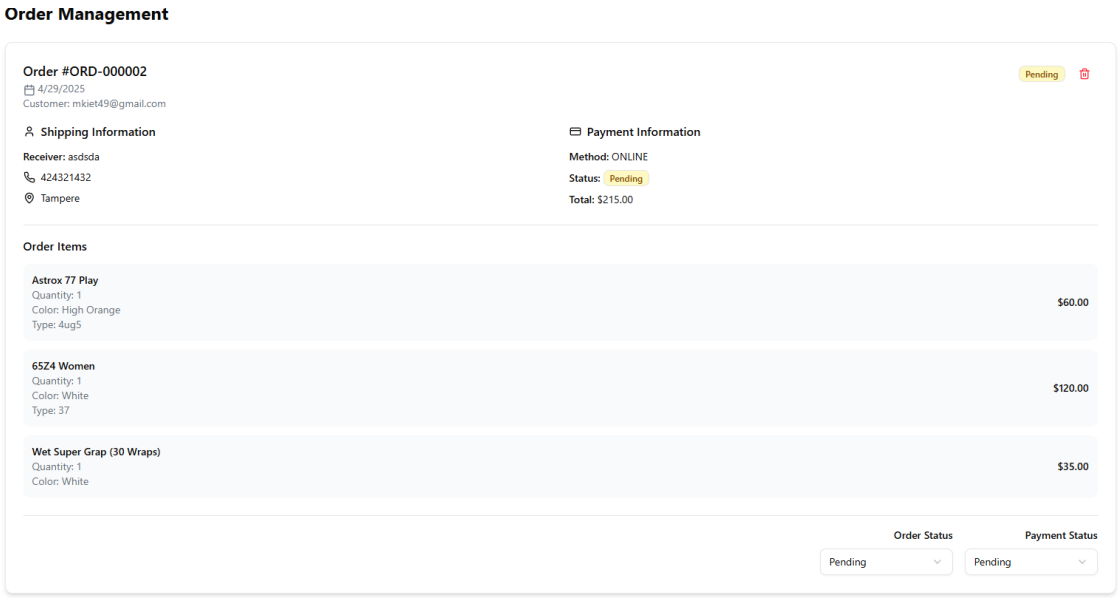


Figure 23. Order management.

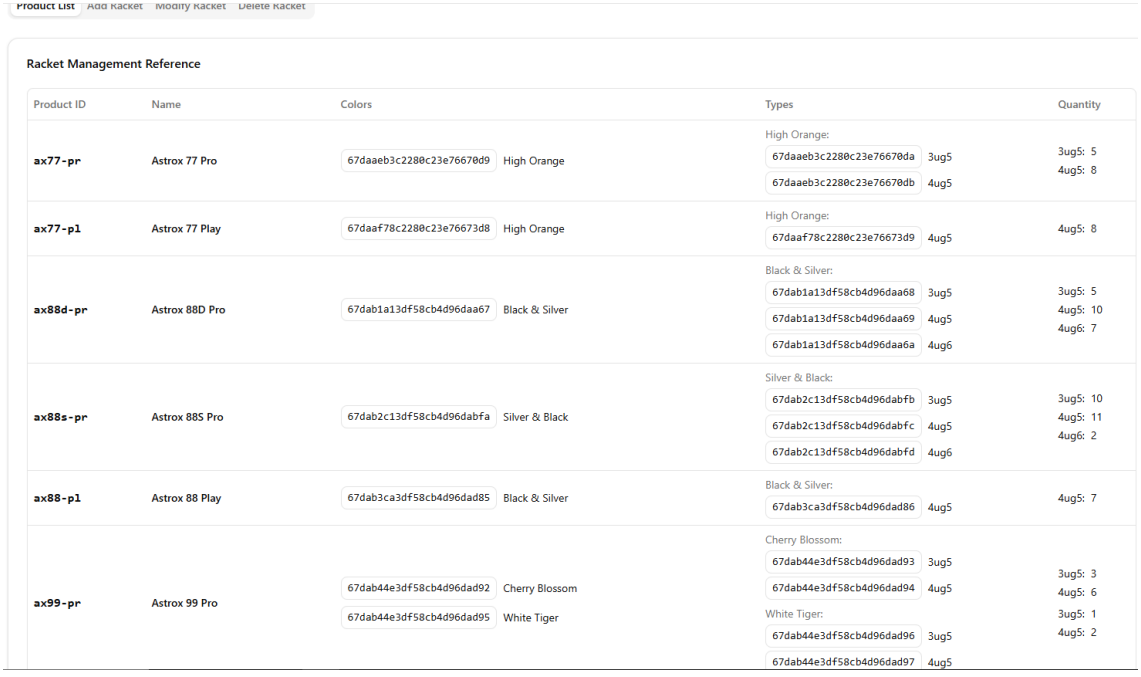


Figure 24. Racket table.

4.3 Differentiation from Existing Solutions

Although a variety of e-commerce platforms exist,such as Shopify, WooCommerce, and Magento. The Badminton Shopping App introduces several distinct features designed to deliver a more specialized and secure online

shopping experience. A primary differentiator is the clear separation between the Customer and Administrator interfaces. This architectural design ensures that each user group operates within an environment specifically tailored to their respective roles. Customers are provided with a streamlined and intuitive interface focused on browsing products, completing purchases, and monitoring order statuses. In contrast, administrators have access to a dedicated management dashboard that facilitates efficient product inventory control, order processing, and delivery status updates.

Additionally, the application supports real-time order status updates that are synchronized between both the customer and administrator views. This feature enhances transparency and builds user trust by ensuring that customers are immediately informed of any changes to their order's status. Unlike many standard e-commerce solutions that require additional paid extensions to enable similar functionality, this feature is integrated by default, offering a more responsive and user-centered experience.

5 CONCLUSIONS AND DISCUSSION

5.1 Outcomes of the development process

The implementation of the e-commerce web application was a clear success in terms of achieving the initial project goals. The final system is functional, coherent, and user-friendly, with a robust set of features tailored for both customers and administrators. The user interface is intuitive and visually consistent, thanks to the integration of React and Tailwind CSS on the frontend. On the backend, the system incorporates secure and scalable practices such as JWT-based authentication and password hashing using bcrypt, as well as RESTful API architecture built with Node.js and Express. MongoDB was chosen as the database for its flexibility and scalability, and TypeScript was used across the entire codebase to improve code safety and maintainability.

The project was developed incrementally, starting with foundational elements such as data models, routing structure, and authentication logic, followed by feature implementation and styling. The clear division of work between the two team members allowed for a parallel workflow, improving both development speed and code quality. Regular integration and code reviews further ensured consistency and minimized integration conflicts.

5.2 Ethical and Practical Considerations

Ethical considerations were also part of the design mindset, especially regarding user data protection. By following common security practices such as encrypted password storage, HTTPS-only communication, and limited data collection, the system aims to comply with user privacy expectations. However, a real-world deployment would require a formal GDPR policy, cookie management, and user rights tools (e.g., data export or deletion requests), which remain as future work. Accessibility and inclusivity are additional aspects that were partly addressed but could be further improved. Future updates should involve testing with screen readers, keyboard navigation, and color contrast checks to ensure the application meets WCAG accessibility standards.

5.3 Future Development Ideas

Looking forward, several enhancements and features could be added to improve the platform:

- Product Recommendation Engine: Implementing machine learning or rule-based product recommendations based on user activity and purchase history.
- User Profile Dashboard: Providing customers with a personal dashboard to track orders, wishlists, and saved items.
- Admin Analytics Panel: Adding an analytics dashboard for administrators to monitor product performance, sales trends, and user activity.
- Inventory Alert System: Automatically notify admins when stock is low to streamline restocking.
- Mobile Responsiveness and PWA Features: Although responsive design was partially implemented, making the app a full Progressive Web App (PWA) would improve mobile usability and offline access.
- Internationalization (i18n): Adding support for multiple languages and currency formats would make the application suitable for a global user base.

5.4 Final Thoughts

To conclude, the project fulfilled its main goal of building a secure, usable, and maintainable e-commerce web application. The technology stack proved to be effective, and the practical experience gained through the project provided valuable insights into real-world full-stack development. While there are areas for further improvement, particularly in testing, accessibility, and long-term scalability, the current version offers a solid foundation for future development or deployment. The lessons learned from both a technical and collaborative perspective will be directly transferable to future software projects and professional practice.

REFERENCES

Arias. (2021). *Hashing in Action: Understanding bcrypt*. Auth0. Retrieved 20.05.2025. https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

Bello. (2023). *What is JWT?* Postman. Retrieved 8.4.2025. https://blog.postman.com/what-is-jwt/

Buchanan. (2025). *Bcrypt*. Asecuritysite. Retrieved 20.5.2025. https://asecuritysite.com/encryption/bcrypt

Chernetskiy. (2024). *Comparing SQL and NoSQL Solutions*. Broscorp. Retrieved 8.4.2025. https://broscorp.net/comparing-sql-and-nosql-solutions/

Cloud.google. (n.d). What is NoSQL database? Retrieved 8.4.2025. https://cloud.google.com/discover/what-is-nosql

Demkovych. (2023). *Monolith vs. Microservices vs. Serverless: Which Architecture is the Best Choice for Your Business?* Geniusee. Retrieved 20.5.2025. https://geniusee.com/single-blog/monolith-vs-microservices-vs-serverless-architecture

DellTechnologies. (n.d.). *MongoDB architecture.* Retrieved 8.4.2025. https://infohub.delltechnologies.com/nl-nl/l/mongodb-on-dell-powerflex-with-nvme-over-tcp-1/mongodb-architecture-3/

GeeksforGeeks. (2024). *E-commerce Design*. Retrieved 5.3.2025. https://www.geeksforgeeks.org/e-commerce-design/

GeeksforGeeks. (2025). *What are the advantages of React.js?* Retrieved 21.5.2025 https://www.geeksforgeeks.org/what-are-the-advantages-of-react-js/

Hashemi-Pour. (2023). *What is e-commerce? TechTarget*. Retrieved 8.4.2025. https://www.techtarget.com/searchcio/definition/e-commerce

Ibrahim. (2024). *What is a JWT? Understanding JSON Web Tokens.* SuperTokens. Retrieved 8.4.2025. https://supertokens.com/blog/what-is-jwt

InfoStride. (2024). *Understanding Stripe Business and Revenue Models.* Retrieved 8.4.2025. https://infostride.com/stripe-business-and-revenue-models/

Kinsta. (2023). *What Is Node.js and Why You Should Use It.* Retrieved 8.4.2025. https://kinsta.com/knowledgebase/what-is-node-js/

Nodejs.org. (n.d). *Introduction to Node.js.* Retrieved 8.4.2025. https://nodejs.org/en/learn/getting-started/introduction-to-nodejs

OECD. (2019). *How's Life in the Digital Age?* Retrieved 20.5.2025. https://www.oecd.org/en/publications/how-s-life-in-the-digital-age_9789264311800-en.html%20%20

Panchal. (2024). *What is ExpressJS in Node JS? – Backend Framework for Web Apps*. ExcellentWebWorld. Retrieved 8.4.2025. https://www.excellentwebworld.com/what-is-expressjs-in-node-js/

Payn. (2025). *What is Stripe?* FreshBooks. Retrieved 8.4.2025. https://www.freshbooks.com/hub/payments/what-is-stripe

Rachowicz. (2024). *When, How, And Why Use Node.js as Your Backend?* Netguru. Retrieved 22.5.2025. https://www.netguru.com/blog/node-js-backend

Reactjs.org. (n.d). *React*. Retrieved 8.4.2025. https://legacy.reactjs.org/

Roy. (2022). *What is Tailwind CSS? A Beginner's Guide*. FreeCodeCamp. Retrieved 6.4.2025. https://www.freecodecamp.org/news/what-is-tailwind-css-a-beginners-guide/

Sharma. (2024). *What is Express JS?* SimpliLearn. Retrieved 8.4.2025. https://www.simplilearn.com/tutorials/nodejs-tutorial/nodejs-express?source=sl_frs_nav_playlist_video_clicked

Singh. (2024). *What is NPM? The best Node JS Package.* MilesWeb. Retrieved 8.4.2025. https://www.milesweb.com/blog/technology-hub/what-is-npm/

Swain. (2025). *What is MERN stack Technology?* Edureka. Retrieved 7.4.2025. https://www.edureka.co/blog/what-is-mern-stack/

Tomar. (2025). *What is TypeScript? A Beginner's Guide.* Guvi. Retrieved 22.05.2025. https://www.guvi.in/blog/what-is-typescript/

Tully. (2025). *Challenges for E-commerce in 2025*. Pimberly. Retrieved 7.3.2025. https://pimberly.com/blog/10-challenges-for-ecommerce-in-2025/

Typescriptlang.org. (n.d). *TypeScript*. Retrieved 8.4.2025. https://www.typescriptlang.org/

Valerie. (2024). *Tailwind CSS Is So Much More Than Just Inline CSS*. Medium. Retrieved 20.05.2025. https://medium.com/dare-to-be-better/tailwind-css-is-so-much-more-than-just-inline-css-plus-its-free-cd70d8347d2c

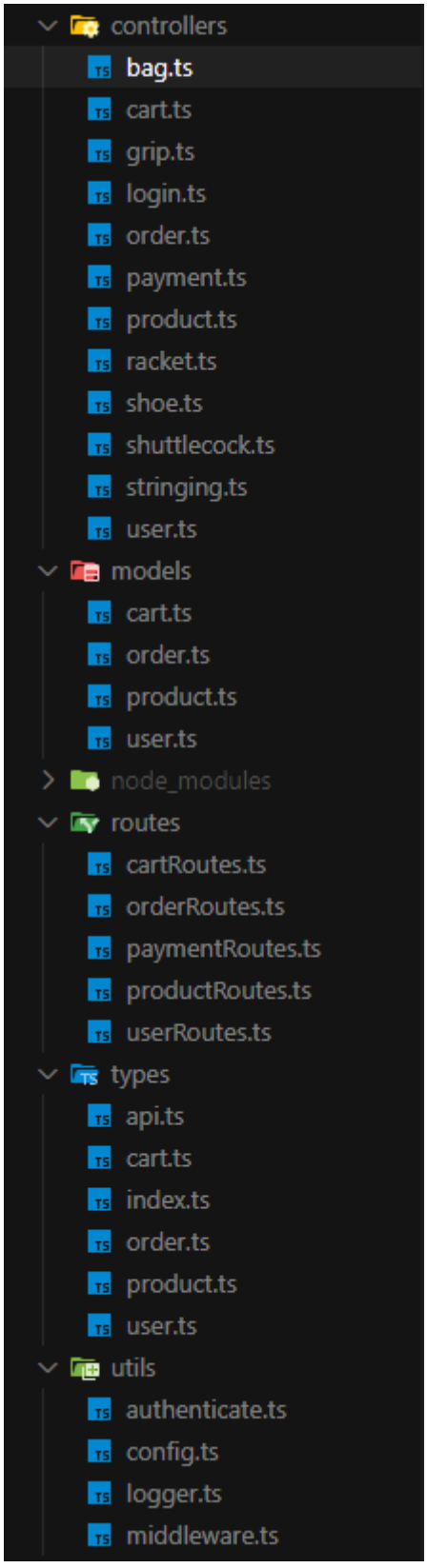
Varghese. (2025). *E-commerce Security*. Astra. Retrieved 10.3.2025. https://www.getastra.com/blog/knowledge-base/ecommerce-security/

W3schools. (n.d). *JavaScript History.* Retrieved 2.3.2025. https://www.w3schools.com/js/js_history.asp

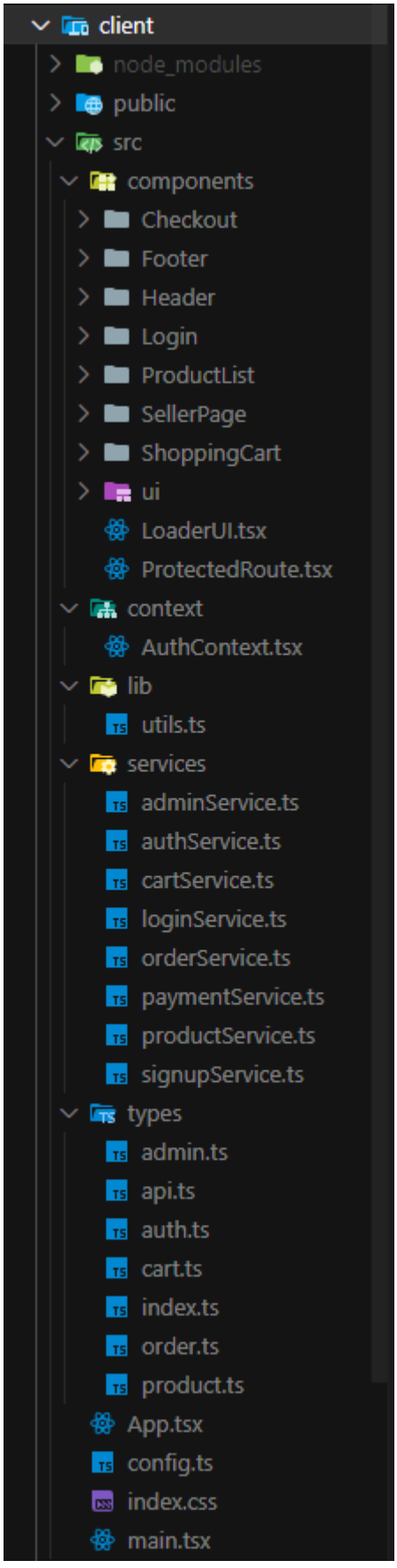
Zhang. (2025). *What is MongoDB? Key Concepts, Use Cases, and Best Practices*. Datacamp. Retrieved 23.05.2025. https://www.datacamp.com/blog/what-is-mongodb

APPENDICES

Appendix 1. Back End Folder Structure.



Appendix 2. Front End Folder Structure.

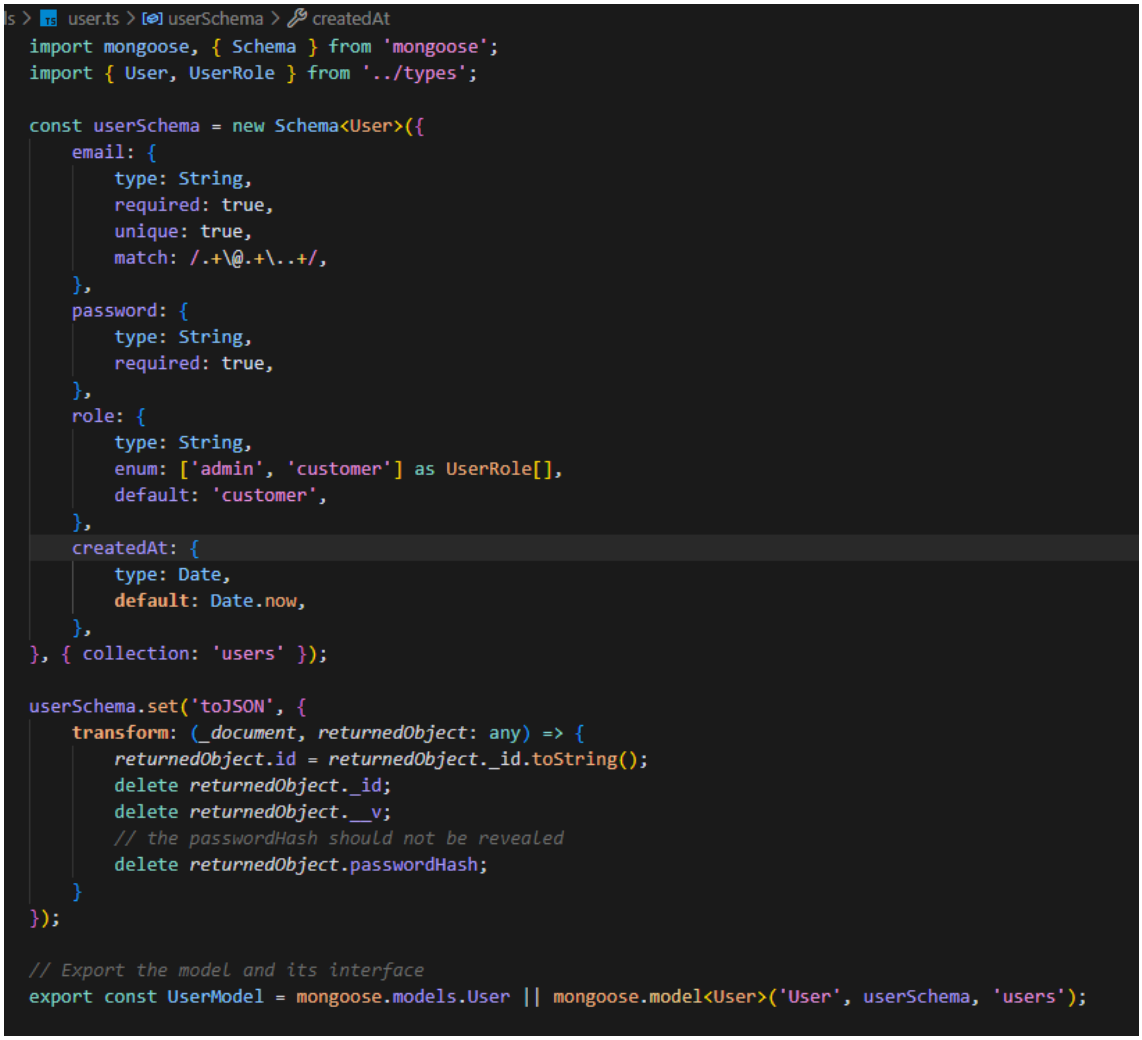


Appendix 3. Authenticate.ts code for authentication.

authenticate.ts > default

```ts
// Function to extract token from the Authorization header
const getTokenFrom = (request: Request): string | null => {
    const authorization = request.header('Authorization');
    if (authorization && authorization.startsWith('Bearer ')) {
        return authorization.replace('Bearer ', '');
    }
    return null;
};

export const authenticateUser = (
    req: AuthRequest,
    res: Response,
    next: NextFunction
): Response | void => {
    const token = getTokenFrom(req);
    if (!token) {
        return res.status(401).json({ error: 'Access denied' });
    }

    try {
        const verified = jwt.verify(token, config.JWT_SECRET) as jwt.JwtPayload;
        req.user = verified;
        next();
    } catch (error) {
        return res.status(400).json({ error: 'Invalid token' });
    }
};

// Middleware to check if the user is an admin
export const isAdmin = (
    req: AuthRequest,
    res: Response,
    next: NextFunction
): Response | void => {
    if (req.user && req.user.role === 'admin') {
        next();
    } else {
        return res.status(403).json({ error: 'Access denied. Admins only.' });
    }
};
```

Appendix 4. User.ts schema for the database.



```
ls > user.ts > userSchema > createdAt
import mongoose, { Schema } from 'mongoose';
import { User, UserRole } from '../types';

const userSchema = new Schema<User>({
    email: {
        type: String,
        required: true,
        unique: true,
        match: /.+\@.+\..+/,
    },
    password: {
        type: String,
        required: true,
    },
    role: {
        type: String,
        enum: ['admin', 'customer'] as UserRole[],
        default: 'customer',
    },
    createdAt: {
        type: Date,
        default: Date.now,
    },
}, { collection: 'users' });

userSchema.set('toJSON', {
    transform: (_document, returnedObject: any) => {
        returnedObject.id = returnedObject._id.toString();
        delete returnedObject._id;
        delete returnedObject.__v;
        // the passwordHash should not be revealed
        delete returnedObject.passwordHash;
    }
});

// Export the model and its interface
export const UserModel = mongoose.models.User || mongoose.model<User>('User', userSchema, 'users');
```

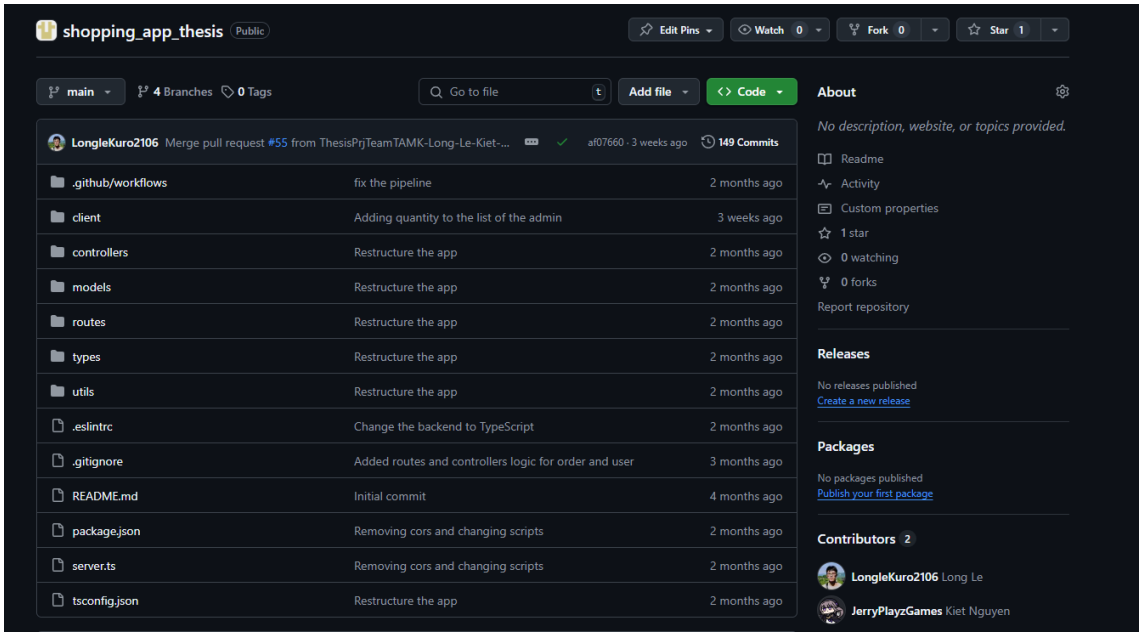
Appendix 5. Payment.ts code for Stripe payment.

```
ntrollers > payment.ts > createPaymentIntent
import { Request, Response } from 'express';
import Stripe from 'stripe';
import config from '../utils/config';
import { OrderModel } from '../models/order';

const stripe = new Stripe(config.STRIPE_SECRET_KEY, {
    apiVersion: '2025-02-24.acacia' // Using the latest API version
});

export const createPaymentIntent = async (req: Request, res: Response) => {
    try {
        const { orderId } = req.body;

        // Get order details
        const order = await OrderModel.findById(orderId);
        if (!order) {
            return res.status(404).json({ success: false, message: 'Order not found' });
        }

        // Create payment intent
        const paymentIntent = await stripe.paymentIntents.create({
            amount: Math.round(order.totalAmount * 100), // Convert to cents
            currency: 'usd',
            metadata: {
                orderId: order._id.toString(),
                orderNumber: order.orderNumber
            }
        });

        // Update order with payment intent ID
        order.paymentIntentId = paymentIntent.id;
        await order.save();

        res.json({
            success: true,
            clientSecret: paymentIntent.client_secret
        });
    } catch (error) {
        console.error('Error creating payment intent:', error);
        res.status(500).json({ success: false, message: 'Error creating payment intent' });
    }
};
```

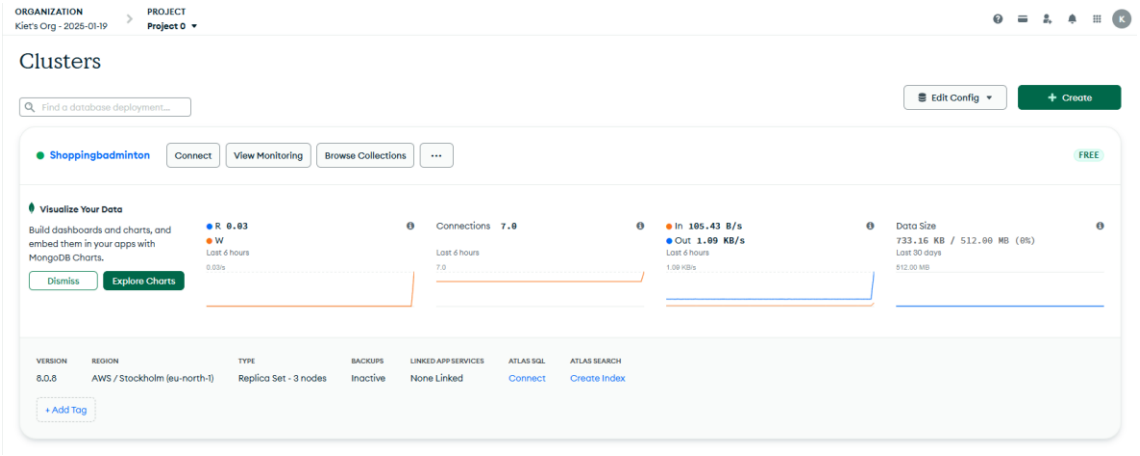
```
export const handleWebhook = async (req: Request, res: Response) => {
    const sig = req.headers['stripe-signature'];
    let event: Stripe.Event;

    try {
        event = stripe.webhooks.constructEvent(
            req.body,
            sig as string,
            config.STRIPE_WEBHOOK_SECRET
        );
    } catch (err: any) {
        console.error('Webhook signature verification failed:', err);
        return res.status(400).send(`Webhook Error: ${err.message}`);
    }

    try {
        switch (event.type) {
            case 'payment_intent.succeeded':
                const paymentIntent = event.data.object as Stripe.PaymentIntent;
                const orderId = paymentIntent.metadata.orderId;

                // Update order status
                await OrderModel.findByIdAndUpdate(orderId, {
                    paymentStatus: 'paid',
                    orderStatus: 'processing'
                });

                break;

            case 'payment_intent.payment_failed':
                const failedPayment = event.data.object as Stripe.PaymentIntent;
                const failedOrderId = failedPayment.metadata.orderId;

                // Update order status
                await OrderModel.findByIdAndUpdate(failedOrderId, {
                    paymentStatus: 'failed'
                });

                break;
        }

        res.json({ received: true });
    } catch (error) {
        console.error('Error handling webhook:', error);
```

```
88              res.status(500).json({ success: false, message: 'Error handling webhook' });
89          }
90      };
```

Appendix 6. GitHub Repository.



Appendix 7. MongoDB Clusters.



Appendix 8. Stripe Dashboard.