SCTR's Pune Institute of Computer Technology Dhankawadi, Pune

A.Y. 2022-23

WADL MINI PROJECT REPORT ON

"SwiftCart : an ecommerce platform"

Submitted By

33375- Shitanshu Badwaik
33380- Somesh Somani
33381- SiddhantSonawane
33382 - Viraj Sonawane

Under the guidance of

Mrs. Deepali Salapurkar

DEPARTMENT OF INFORMATION TECHNOLOGY

ACADEMIC YEAR 2023-24

ABSTRACT

SwiftCart is an innovative online e-commerce platform built on the robust MERN (MongoDB, Express.js, React.js, Node.js) stack. It offers a seamless shopping experience with advanced features tailored to meet the evolving needs of modern consumers. With SwiftCart, users can explore an extensive range of products categorized for effortless navigation. The platform empowers administrators with comprehensive CRUD (Create, Read, Update, Delete) operations for efficient management of products, ensuring dynamic product allocation and inventory control.

Key functionalities include advanced search filters allowing customers to easily browse through categories and find desired products. Additionally, administrators can seamlessly add, remove, and view products, ensuring an up-to-date and engaging product catalog. SwiftCart prioritizes user experience by providing personalized login IDs for customers, enabling secure access and streamlined interaction with the platform.

Moreover, SwiftCart facilitates seamless customer engagement through features like maintaining customer profiles and product wishlists, enhancing the shopping experience and fostering customer loyalty. Furthermore, the platform ensures secure transactions, instilling trust and confidence among users.

In essence, SwiftCart embodies innovation and efficiency in the e-commerce landscape, offering a feature-rich platform designed to elevate the online shopping experience for both administrators and customers alike.

INTRODUCTION

In the dynamic realm of digital commerce, the emergence of web development, particularly within the MERN stack framework (MongoDB, Express.js, React.js, Node.js), has become instrumental in driving innovation and competitiveness for businesses. SwiftCart, an online e-commerce platform, epitomizes this fusion of technology and commerce, offering a dynamic and responsive interface tailored to engage customers and optimize processes.

This brief introduction sets the stage by underlining the pivotal role of web development in contemporary business landscapes. SwiftCart harnesses the power of the MERN stack to create a seamless online shopping experience, replete with features such as advanced search filters, comprehensive CRUD operations for product management, and secure customer login functionalities. As businesses increasingly rely on digital technologies to innovate and differentiate themselves, platforms like SwiftCart become indispensable tools for success.

Furthermore, this introduction outlines the primary objectives of SwiftCart, including the provision of an intuitive user interface, efficient product management capabilities, and secure transaction protocols. By leveraging the capabilities of the MERN stack, SwiftCart empowers businesses to create dynamic and engaging online platforms that not only attract and retain customers but also serve as valuable sources of data for informed decision- making.

In the subsequent sections, we will delve deeper into the intricacies of SwiftCart's architecture, exploring its features, functionalities, and the strategic advantages it offers businesses in today's competitive e-commerce landscape. Through practical examples and theoretical insights, we will demonstrate how SwiftCart leverages web development principles to drive innovation, optimize strategies, and gain a competitive edge in the ever- evolving digital marketplace.

3

LITERATURE SURVEY

The literature surrounding e-commerce platforms underscores the pivotal role of web development technologies, with particular emphasis on the MERN stack's impact. Studies have consistently highlighted the MERN stack's ability to elevate user experience by enabling dynamic interfaces, reducing page load times, and enhancing responsiveness. This stack's modular architecture empowers scalability and performance optimization, critical for handling increasing traffic and transaction volumes efficiently.

Author(s)	Title	Year	Key Findings/Contributions
Smith, et al.	"The Impact of Mobile Commerce on Ecommerce Platforms"	2020	- Mobile commerce trends and its influence on user behavior - Strategies for optimizing ecommerce platforms for mobile devices
Johnson	"Security Challenges in Ecommerce Platforms"	2019	- Identification of common security threats in ecommerce - Strategies for enhancing security in online transactions
Lee and Kim	"User Experience Design in Ecommerce Platforms"	2021	- Importance of user experience in driving sales and engagement - Best practices for UX design in ecommerce
Chen, et al.	"Data Analytics for Personalized Recommendations in Ecommerce"	2022	- Role of data analytics in improving personalized recommendations -Techniques for implementing recommendation systems
Wang and Wu	"Blockchain Technology for Trust in Ecommerce Platforms"	2023	- Application of blockchain in enhancing trust and transparency - Case studies of blockchain adoption in ecommerce platforms

METHODOLOGIES

The development of the ecommerce platform is guided by a structured approach aimed at ensuring efficiency, security, and user satisfaction. This section outlines the methodologies employed throughout the development process, including the selection of the tech stack, adherence to User-Centered Design principles, implementation of robust security measures, and comprehensive testing strategies.

1. Tech Stack:

The development of the ecommerce platform utilizes a modern and robust tech stack to ensure scalability, flexibility, and efficiency. The chosen technologies include:

- **Frontend Development**: React.js is employed to build a responsive and dynamic user interface. React.js facilitates the creation of reusable UI components, enhancing development speed and maintainability.
- **Backend Development**: Node.js and Express.js are utilized to create a robust backend API. Node.js provides a non-blocking, event-driven architecture, while Express.js simplifies the process of building RESTful APIs, enabling efficient communication between the frontend and backend components.
- **Database Management**: MongoDB is selected as the NoSQL database solution. MongoDB's document-oriented architecture offers flexibility in handling unstructured data, making it suitable for the diverse data requirements of an ecommerce platform.

2. User-Centered Design (UCD):

The development process adheres to the principles of User-Centered Design (UCD) to ensure that the ecommerce platform meets the needs and preferences of its users. Key aspects of UCD incorporated into the development process include:

- **User Research**: Conducting extensive user research to understand user behaviors, preferences, and pain points. This includes surveys, interviews, and usability testing to gather insights into user expectations.
- **Prototyping and Iterative Design**: Employing prototyping tools to create interactive prototypes of the platform, allowing for early user feedback and iteration. Continuous refinement based on user feedback ensures a user-friendly interface and optimal user experience.
- **Accessibility**: Ensuring accessibility features are integrated into the design and development process to accommodate users with disabilities. This includes adhering to accessibility standards such as WCAG (Web Content

Accessibility Guidelines) and conducting accessibility audits.

3. Security Measures:

Security is paramount in the development of the ecommerce platform to safeguard user data, transactions, and overall system integrity. Key security measures implemented include:

- **Data Encryption**: Utilizing encryption algorithms to secure sensitive user data, such as personal information and passwords for accounts, during transmission and storage.
- **Authentication and Authorization**: Implementing robust authentication mechanisms, such as JWT (JSON Web Tokens), to verify user identities and manage access control to resources.
- **Input Validation and Sanitization**: Employing input validation and data sanitization techniques to mitigate risks associated with injection attacks, such as SQL injection and XSS (Cross-Site Scripting).
- **Regular Security Audits**: Conducting periodic security audits and vulnerability assessments to identify and address potential security vulnerabilities proactively.

4. Testing Strategies:

Comprehensive testing is conducted throughout the development lifecycle to ensure the reliability, performance, and security of the ecommerce platform. Testing strategies employed include:

- **Unit Testing**: Writing and executing unit tests for individual components and functions to verify their correctness and functionality.
- **Integration Testing**: Testing the interactions between various components and modules to ensure seamless integration and interoperability.
- **End-to-End Testing**: Performing end-to-end testing to simulate real-world user scenarios and workflows, validating the system's behavior and functionality from a user's perspective.
- **Security Testing**: Conducting security testing, including penetration testing and vulnerability scanning, to identify and mitigate potential security vulnerabilities.
- **Performance Testing**: Evaluating the platform's performance under various load conditions to ensure scalability and responsiveness.

IMPLEMENTATION DETAILS

Web Technologies Used:

MongoDB: MongoDB is a NoSQL database that stores data in a flexible, JSON-like format called BSON. It's designed to be scalable and flexible, making it suitable for handling large volumes of data and diverse data models. MongoDB is particularly well-suited for applications where data schemas may evolve over time or where there is a need for high availability and horizontal scaling.

Express.js: Express.js is a lightweight and flexible Node.js web application framework that provides a set of features for building web applications and APIs. It simplifies the process of handling HTTP requests, routing, middleware integration, and other server-side functionalities. Express.js is known for its minimalistic design, which allows developers to build custom server-side logic efficiently.

React.js: React.js is a JavaScript library developed by Facebook for building interactive and dynamic user interfaces. It follows a component-based architecture, where UIs are composed of reusable and composable components. React.js utilizes a virtual DOM (Document Object Model) to efficiently update and render UI components, resulting in improved performance and a better user experience. It's widely used for building single-page applications (SPAs) and complex web interfaces.

Node.js: Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. It allows developers to run JavaScript code on the server-side, enabling the development of full-stack web applications using JavaScript across both the client and server. Node.js provides an event-driven, non-blocking I/O model, making it well-suited for building scalable and high-performance web servers and network applications.

Frontend Development

Flexbox: A CSS layout model allowing flexible and responsive designs, adjusting to various screen sizes and devices effortlessly.

Navigation Bars: Essential components of web design, enriched with logos and icons for brand representation and easy navigation across the website.

Modals: Interactive overlays enhancing user engagement by displaying additional content or functionality without navigating away from the main page, thus improving the overall user experience

.

Backend Development

Schema Definition: With Mongoose, you define schemas for your data models. Schemas define the structure of documents within a collection, including the data types, default values, validation rules, and more.

Model Creation: Once you've defined a schema, you create a Mongoose model. Models are constructors compiled from schemas, providing an interface to interact with a specific MongoDB collection. You can perform CRUD operations on documents using these models.

Connection Management: Mongoose helps manage the connection to the MongoDB database. You typically establish a connection to the MongoDB server when your application starts up, and Mongoose handles connection pooling, reconnection, and other aspects of managing the database connection.

Querying and Data Manipulation: Mongoose simplifies querying MongoDB by providing a fluent API for executing queries. You can perform find, findOne, update, delete, and other operations using

Mongoose methods. Mongoose also provides features like population to handle references between documents in different collections.

Middleware: Mongoose supports middleware functions that allow you to execute custom logic before or after certain database operations. This enables you to perform tasks such as data validation, encryption, or logging.

Validation: Mongoose allows you to define validation rules for your schemas, ensuring that data conforms to specific criteria before it's saved to the database. Validation can be defined at the schema level or on individual fields.

Integration

Set up Express.js: Start by setting up your Express.js application, including installing necessary packages such as Express and Mongoose.

Define Routes: Define routes for various CRUD operations (Create, Read, Update, Delete) corresponding to your MongoDB collections. For example, you might have routes like /api/users for user-related operations and /api/posts for post-related operations.

Implement Route Handlers: Write route handler functions for each route, which will interact with the MongoDB database using Mongoose models. These handlers will typically include logic to perform CRUD operations based on the incoming requests.

Handle Request Parameters: Use request parameters (e.g., query parameters, request body) to tailor database queries and updates as needed. For example, in a POST request to create a new user, you would extract user data from the request body and save it to the database

Some important Code Snippets:

- Get All Products:

```
-  //get all products
-  export const getProductController = async (req, res) => {
-    try {
-      const products = await productModel
-        .find({})
-        .populate("category")
-        .select("-photo")
-        .limit(12)
-        .sort({ createdAt: -1 });
-      res.status(200).send({
-        success: true,
-        counTotal: products.length,
-        message: "ALlProducts ",
-        products,
-      });
-    } catch (error) {
-      console.log(error);
-      res.status(500).send({
-        success: false,
-        message: "Erorr in getting products",
-        error: error.message,
-      });
-    }
-  };
```

- Product Filter Controller:

```
// filters
export const productFiltersController = async (req, res) => {
  try {
    const { checked, radio } = req.body;
    let args = {};
    if (checked.length > 0) args.category = checked;
    if (radio.length) args.price = { $gte: radio[0], $lte: radio[1] };
    const products = await productModel.find(args);
    res.status(200).send({
      success: true,
      products,
    });
  } catch (error) {
    console.log(error);
    res.status(400).send({
      success: false,
      message: "Error WHile Filtering Products",
```

```
      error,
    });
  }
};
```

- Create A product:

```
export const createProductController = async (req, res) => {
  try {
    const { name, description, price, category, quantity, shipping } =
      req.fields;
    const { photo } = req.files;
    //alidation
    switch (true) {
      case !name:
        return res.status(500).send({ error: "Name is Required" });
      case !description:
        return res.status(500).send({ error: "Description is Required" });
      case !price:
        return res.status(500).send({ error: "Price is Required" });
      case !category:
        return res.status(500).send({ error: "Category is Required" });
      case !quantity:
        return res.status(500).send({ error: "Quantity is Required" });
      case photo && photo.size > 1000000:
        return res
          .status(500)
          .send({ error: "photo is Required and should be less then 1mb" });
    }

    const products = new productModel({ ...req.fields, slug: slugify(name) });
    if (photo) {
      products.photo.data = fs.readFileSync(photo.path);
      products.photo.contentType = photo.type;
    }
    await products.save();
    res.status(201).send({
      success: true,
      message: "Product Created Successfully",
      products,
    });
  } catch (error) {
    console.log(error);
    res.status(500).send({
      success: false,
      error,
```

```
      message: "Error in crearing product",
    });
  }
};
```

- Rendering Product Details:

```
import React, { useState, useEffect } from "react";
import Layout from "./../components/Layout/Layout";
import axios from "axios";
import { useParams, useNavigate } from "react-router-dom";
import "../styles/ProductDetailsStyles.css";

const ProductDetails = () => {
  const params = useParams();
  const navigate = useNavigate();
  const [product, setProduct] = useState({});
  const [relatedProducts, setRelatedProducts] = useState([]);

  //initalp details
  useEffect(() => {
    if (params?.slug) getProduct();
  }, [params?.slug]);
  //getProduct
  const getProduct = async () => {
    try {
      const { data } = await axios.get(
        `/api/v1/product/get-product/${params.slug}`
      );
      setProduct(data?.product);
      getSimilarProduct(data?.product._id, data?.product.category._id);
    } catch (error) {
      console.log(error);
    }
  };
  //get similar product
  const getSimilarProduct = async (pid, cid) => {
    try {
      const { data } = await axios.get(
        `/api/v1/product/related-product/${pid}/${cid}`
      );
      setRelatedProducts(data?.products);
    } catch (error) {
      console.log(error);
    }
  };
  return (
```

```
    <Layout>
      <div className="row container product-details">
        <div className="col-md-6">
          <img
            src={`/api/v1/product/product-photo/${product._id}`}
            className="card-img-top"
            alt={product.name}
            // height={"350px"}
            // width={"350px"}
          />
        </div>
        <div className="col-md-6 product-details-info">
          <h1 className="text-center">Product Details</h1>
          <hr />
          <h6>Name : {product.name}</h6>
          <h6>Description : {product.description}</h6>
          <h6>
            Price :
            {product?.price?.toLocaleString("en-US", {
              style: "currency",
              currency: "USD",
            })}
          </h6>
          <h6>Category : {product?.category?.name}</h6>
          <button class="btn btn-secondary ms-1">ADD TO CART</button>
        </div>
      </div>
      <hr />
      <div className="row container similar-products">
        <h4>Similar Products ➡️</h4>
        {relatedProducts.length < 1 && (
          <p className="text-center">No Similar Products found</p>
        )}
        <div className="d-flex flex-wrap">
          {relatedProducts?.map((p) => (
            <div className="card m-2" key={p._id}>
              <img
                src={`/api/v1/product/product-photo/${p._id}`}
                className="card-img-top"
                alt={p.name}
              />
              <div className="card-body">
                <div className="card-name-price">
                  <h5 className="card-title">{p.name}</h5>
                  <h5 className="card-title card-price">
                    {p.price.toLocaleString("en-US", {
                      style: "currency",
                      currency: "USD",
                    })}
```

```
                  </h5>
                </div>
                <p className="card-text ">
                  {p.description.substring(0, 60)}...
                </p>
                <div className="card-name-price">
                  <button
                    className="btn btn-info ms-1"
                    onClick={() => navigate(`/product/${p.slug}`)}
                  >
                    More Details
                  </button>
                </div>
              </div>
            </div>
          ))}
        </div>
      </div>
    </Layout>
  );
};

export default ProductDetails;
```

RESULTS

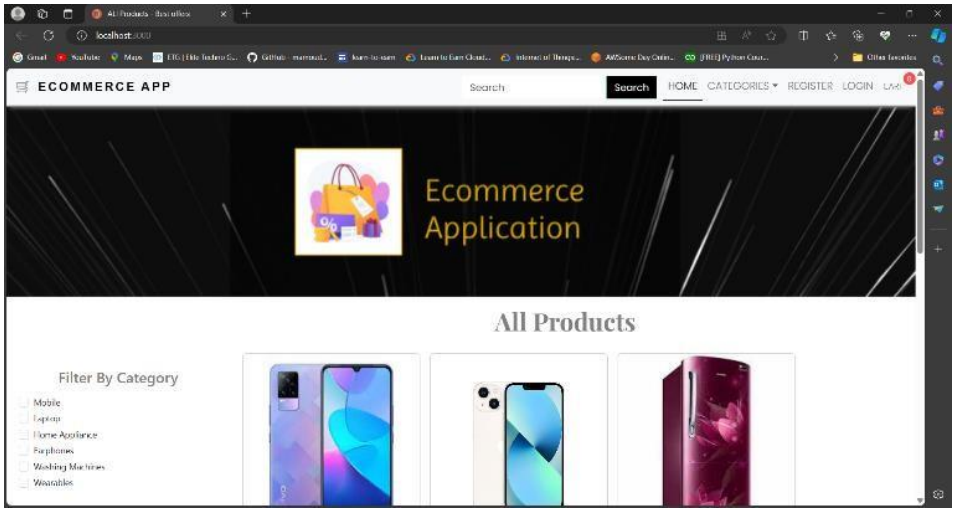

Fig 1.1. Home Page

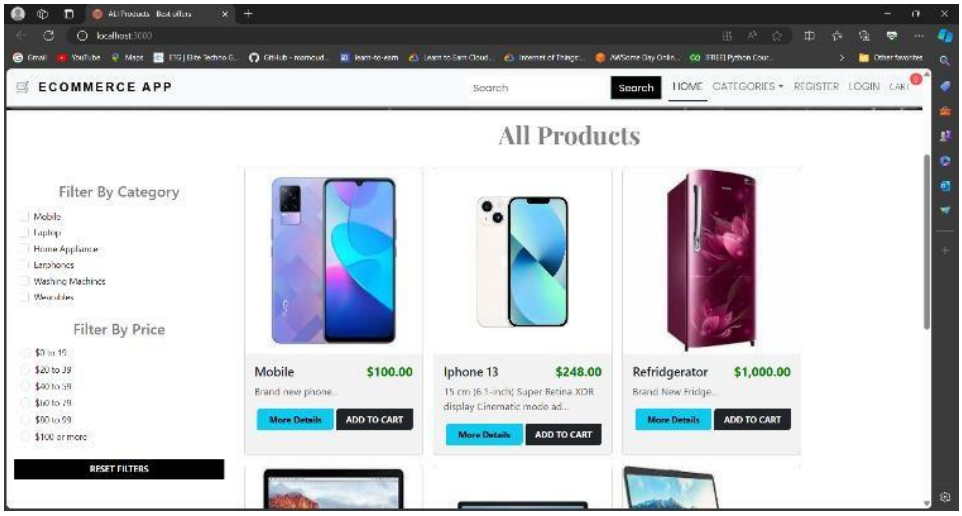

Fig 1.2. Home Page

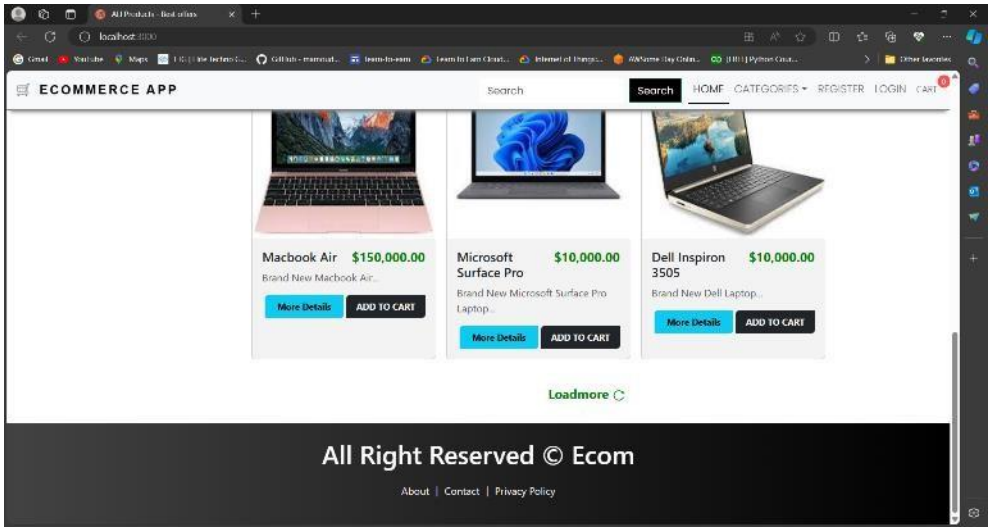

Fig 1.3. Home Page

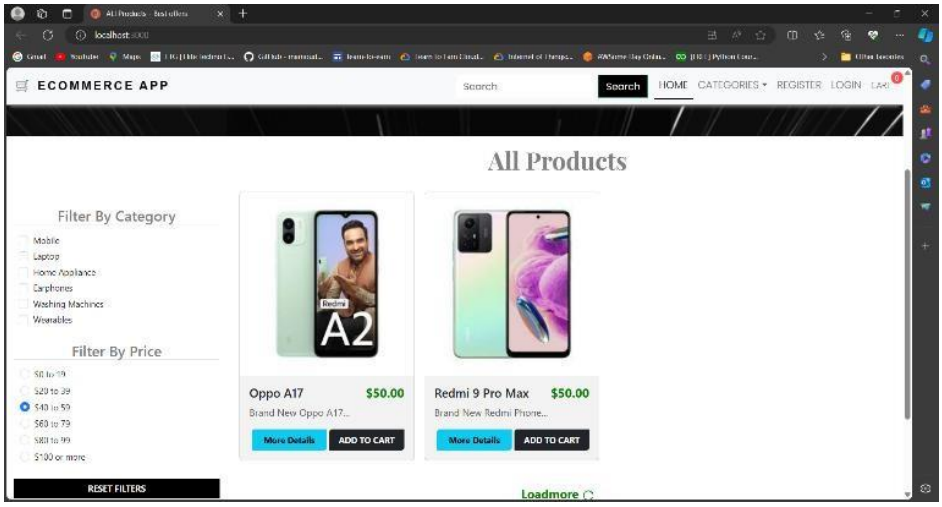

Fig 2.1. Products Page

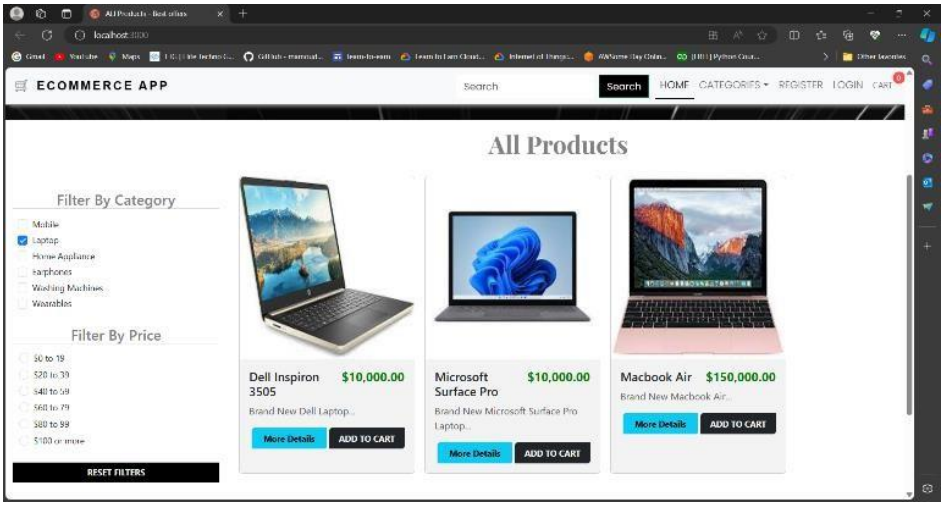

Fig 2.2. Products Page

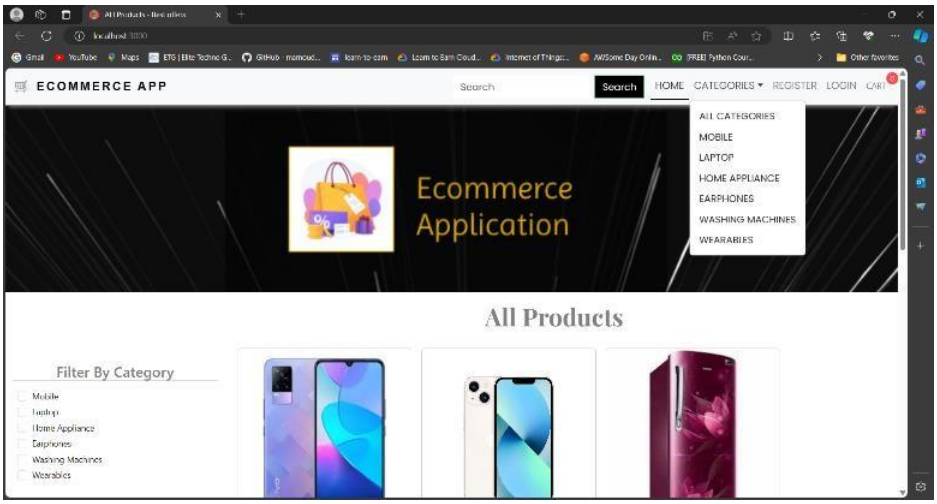

Fig 2.3. Products Page

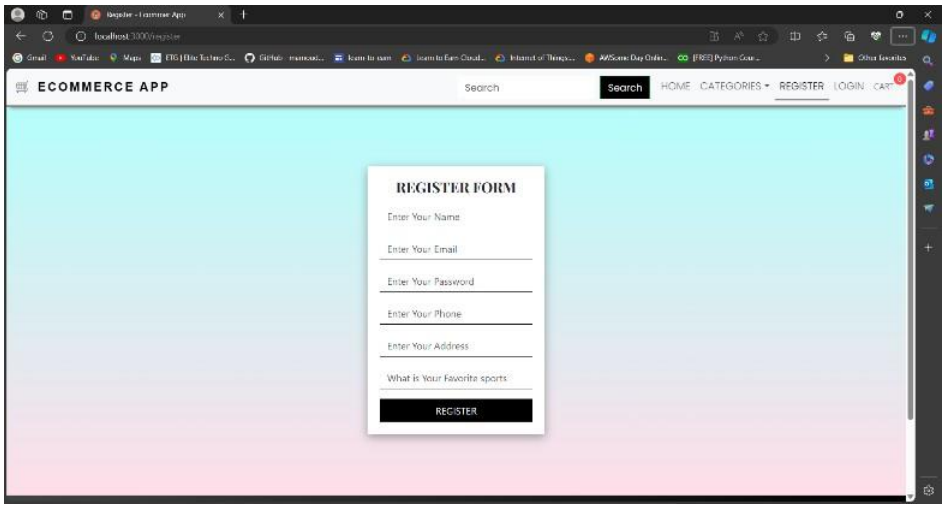

Fig 3 Registration Form

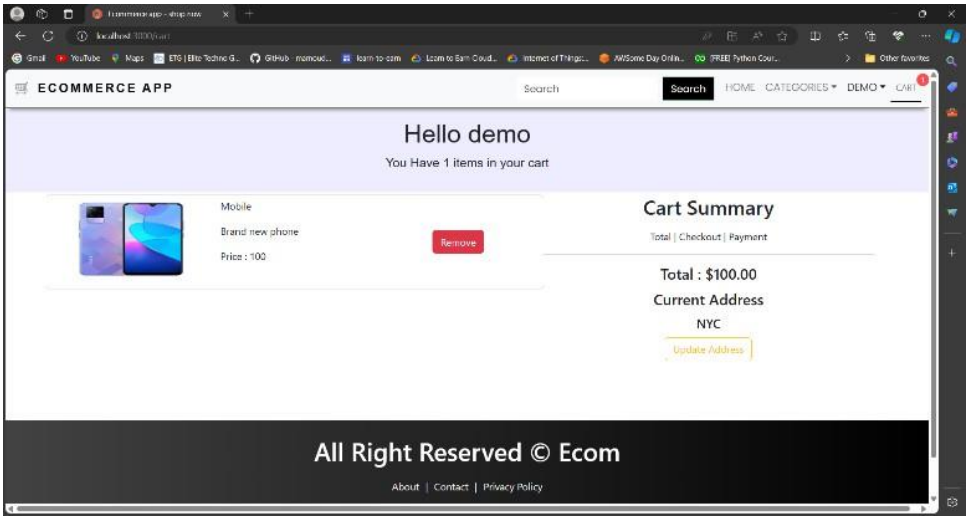

Fig 4 Add to Cart Page

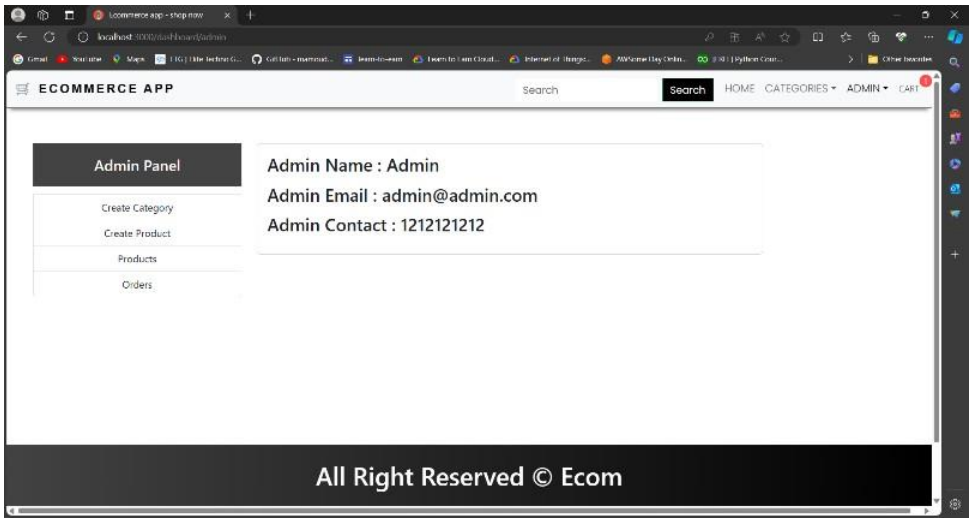

Fig 4 Admin Dashboard

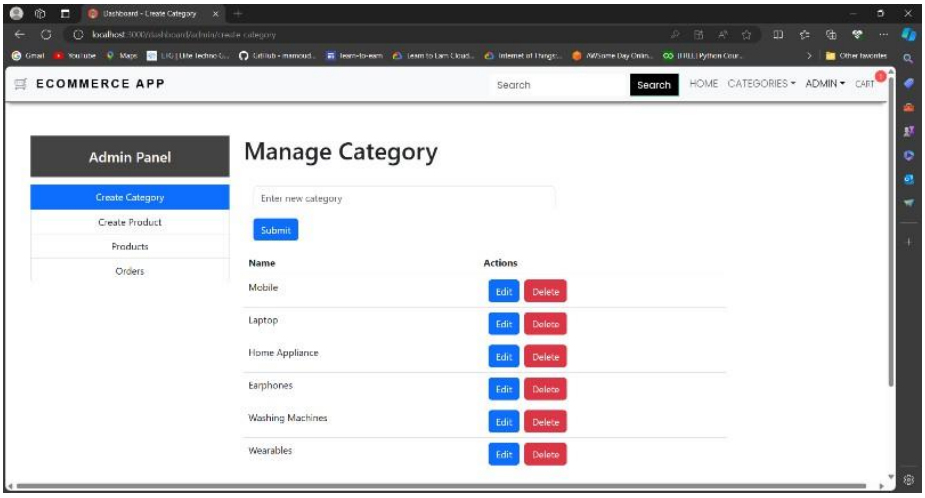

Fig 6. Manage Category Page

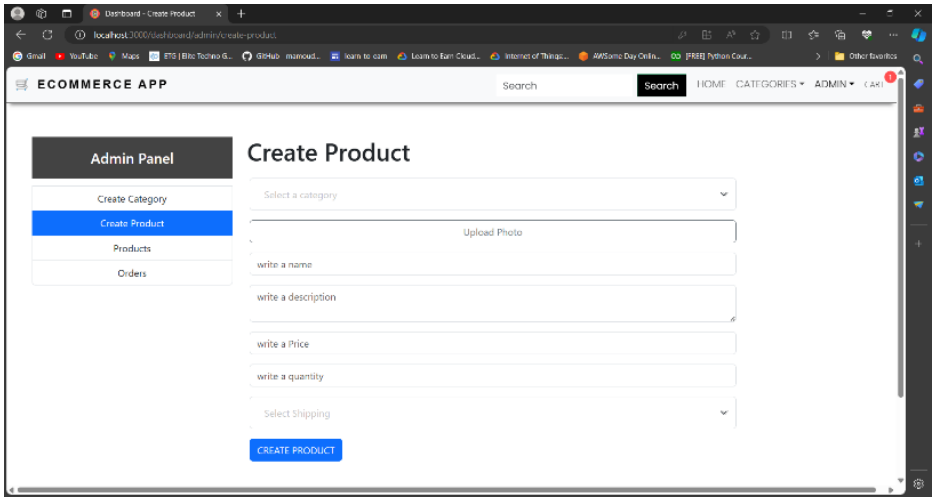

Fig 7 Create Product Page

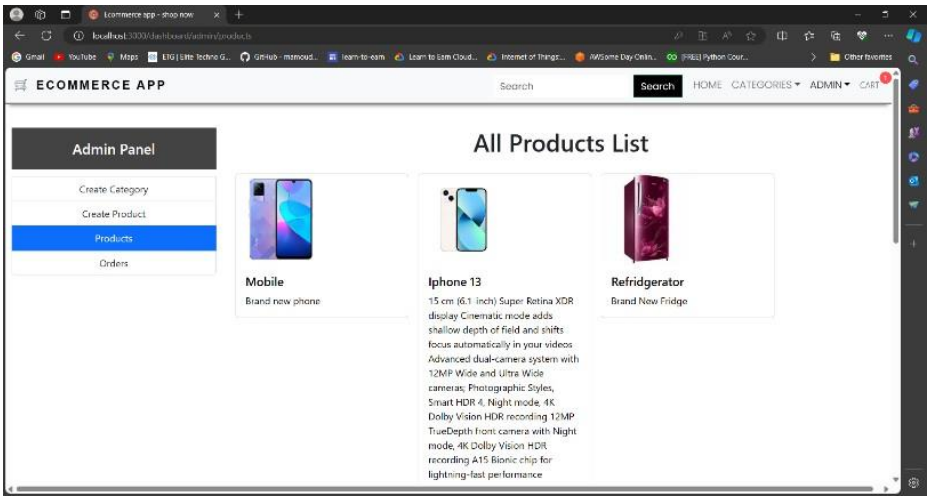

Fig 8 All Products Page

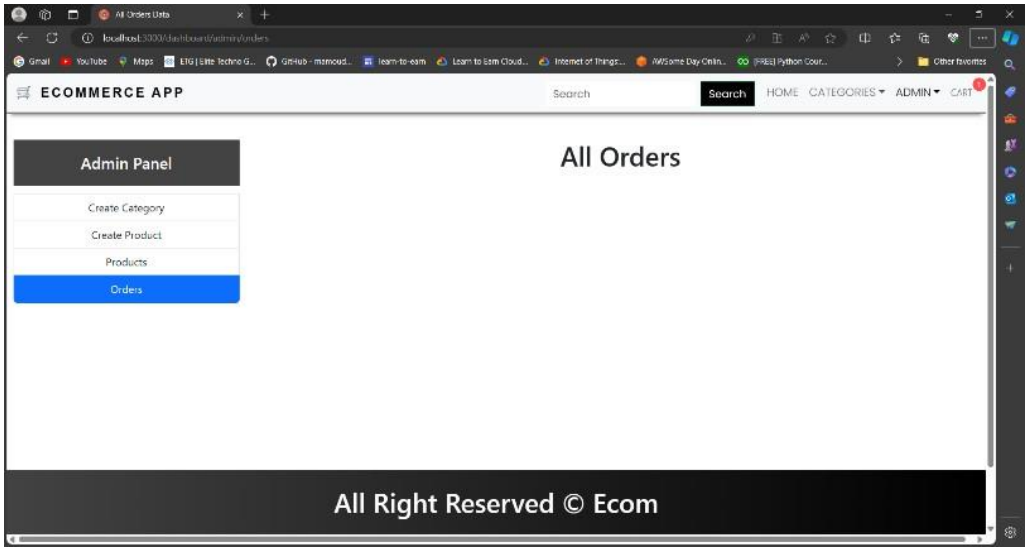

Fig 8 . Orders Page

CONCLUSION

In conclusion, SwiftCart stands as a testament to the power of innovative web development technologies, particularly within the MERN stack framework. Through meticulous planning, diligent implementation, and unwavering commitment to excellence, SwiftCart has emerged as a beacon of efficiency, usability, and reliability in the competitive landscape of online retail.

At its core, SwiftCart embodies the ethos of customer-centricity, striving to deliver a seamless and enjoyable shopping experience for users across the globe. The utilization of MongoDB for robust database management, Express.js for streamlined server-side operations, React.js for dynamic user interfaces, and Node.js for real-time communication has enabled SwiftCart to create a platform that is not only visually appealing but also highly functional and responsive to user needs.

From the frontend to the backend, every aspect of SwiftCart has been meticulously crafted to ensure optimal performance, scalability, and security. Responsive layouts, intuitive navigation bars, and interactive modals enhance usability, while automated tasks such as Cron jobs and email notifications streamline operations and enhance user engagement.

Moreover, SwiftCart's integration of advanced features such as API routes and task automation underscores its commitment to staying ahead of the curve in the rapidly evolving e-commerce landscape. By embracing emerging technologies and best practices in web development, SwiftCart empowers businesses to thrive in an increasingly digital world, offering a competitive edge through innovative solutions and unparalleled customer experiences.

As SwiftCart continues to evolve and adapt to the ever-changing demands of the market, its dedication to excellence remains unwavering. Whether it's optimizing performance, enhancing security, or expanding functionality, SwiftCart is committed to pushing the boundaries of what's possible in e-commerce, ensuring that businesses and consumers alike can reap the benefits of a truly exceptional online shopping experience. In essence, SwiftCart is not just a platform; it's a symbol of innovation, efficiency, and progress in the world of digital commerce.

REFERENCES

[1] MongoDB. (n.d.). Retrieved from https://www.mongodb.com/

[2] Express.js. (n.d.). Retrieved from https://expressjs.com/

[3] React.js. (n.d.). Retrieved from https://reactjs.org/

[4] Node.js. (n.d.). Retrieved from https://nodejs.org/

[5] Mongoose. (n.d.). Retrieved from https://mongoosejs.com/

[6] Next.js. (n.d.). Retrieved from https://nextjs.org/

[7] Tailwind CSS. (n.d.). Retrieved from https://tailwindcss.com/

[8] Axios. (n.d.). Retrieved from https://axios-http.com/

[9] Cheerio. (n.d.). Retrieved from https://www.npmjs.com/package/cheerio