# **CHAPTER 1 INTRODUCTION**

Today's dynamic agricultural sector offers technology and traditional farming practices innovation opportunities. Sustainability, community involvement, and accessibility make systems that connect local farmers with consumers to exchange fresh, locally produced agricultural products attractive.

Traditional agricultural product promotion has limited reach and availability due to community recommendations or farmer markets. The "Agriculture Project Marketplace and Local Product Search" uses innovative technologies and community-driven solutions to improve agriculture within boundaries. Complex web apps will be developed with MERN.

Programme promotes farmer-consumer dialogue. Through digital connectivity, this platform changes agricultural product display, location, and purchasing. Community involvement, sustainable farming, and local farming projects are its key objectives.

In the intricate agricultural trading network, technology enables digital participants. From project inception to launch, we examine the challenges of creating a dynamic, user-focused platform that crosses boundaries, builds meaningful relationships, and supports local agriculture. Discover productive creation and build a clickable universe with us

# **CHAPTER 2 SYSTEM ANALYSIS**

**2.1 EXISTING SYSTEM**

The current agricultural landscape, farmers primarily rely on traditional. Farming practices and local farmer markets to showcase and sell their produce. The exchange of agricultural products often takes place through word of mouth within local communities, and farmers may directly interact with consumers or local businesses.

However, this approach has limitations in terms of visibility, often confined to a specific geographic area. To enhance and modernize these existing methods, the proposed Agriculture Project Marketplace and Local Product Search web application aims to provide a digital platform.

In response to the limitations of traditional agricultural practices, the proposed Agriculture Project Marketplace and Local Product Search system will revolutionize the way farmers showcase and sell their products while providing consumers with a convenient and efficient platform to discover and purchase fresh, locally sourced agricultural items.

Leveraging the MERN stack, this web application will introduce several innovative features. An interactive map will visualize the geographic locations of agricultural projects, offering users a dynamic and immersive experience. Users will have the ability to search for specific agricultural products based on their location or preferences, breaking down geographic constraints.

The system will facilitate communication between farmers and consumers, allowing users to express interest, negotiate terms, and arrange transactions directly within the platform.

Notifications and alerts will keep users informed about project updates and product availability, enhancing engagement. Furthermore, the application will prioritize responsive design, ensuring a seamless experience across various devices.

This proposed system aims to bridge the gap between local farmers and consumers, fostering community interaction, supporting sustainable agriculture, and creating a user-friendly platform that promotes the growth of local farming initiatives. With enhanced security   
measures, thorough testing, and deployment on a reliable cloud platform, the Agriculture Project Marketplace and Local Product Search system seeks to empower both farmers and consumers in a digital age.

2.2 **PROPOSED SYSTEM**

In response to the limitations of traditional agricultural practices, the proposed Agriculture Project Marketplace and Local Product Search system will revolutionize the way farmers showcase and sell their products while providing consumers with a convenient and efficient platform to discover and purchase fresh, locally sourced agricultural items.

Leveraging the MERN stack, this web application will introduce several innovative features. An interactive map will visualize the geographic locations of agricultural projects, offering users a dynamic and immersive experience.

Users will have the ability to search for specific agricultural products based on their location or preferences, breaking down geographic constraints.

The system will facilitate communication between farmers and consumers, allowing users to express interest, negotiate terms, and arrange transactions directly within the platform.

Notifications and alerts will keep users informed about project updates and product availability, enhancing engagement. Furthermore, the application will prioritize responsive design, ensuring a seamless experience across various devices.

This proposed system aims to bridge the gap between local farmers and consumers, fostering community interaction, supporting sustainable agriculture, and creating a user-friendly platform that promotes the growth of local farming initiatives. With enhanced security measures, thorough testing, and deployment on a reliable cloud platform, the Agriculture Project Marketplace and Local Product Search system seeks to empower both farmers and consumers in a digital age.

# **CHAPTER 3 SYSTEM REQUIREMENTS**

* 1. **HARDWARE REQUIREMENTS:**

System : Intel i3 3.0 GHz.

Hard Disk : 1TB.

Monitor : 14’ Colour Monitor.

Ram : 8GB.

* 1. **SOFTWARE REQUIREMENTS:**

Operating system : Windows 10.

Coding Language : JavaScript.

Front-End : React.js.

Back-End : Node.js(Environment)/Express.js .

Data Base : MongoDB.

* 1. **SOFTWARE ENVIRONMENT JAVASCRIP (programming Language)**

JavaScript, often abbreviated as JS, is a high-level, interpreted programming language that is primarily known for its use in web development. Here's a brief history of JavaScript:

**CREATION AT NETSCAPE:**

JavaScript was created by Brendan Eich while he was working at Netscape Communications Corporation. It was originally developed under the name Mocha, which was later renamed to Live Script, and finally, JavaScript. The language was introduced in Netscape Navigator 2.0 in December 1995.

**STANDARDIZATION WITH ECMASCRIPT:**

In November 1996, JavaScript was submitted to ECMA International to carve out a standard specification, leading to the creation of ECMAScript. ECMAScript is the standardized specification upon which JavaScript implementations are based. The first edition of ECMAScript was released in June 1997.

**BROWSER WARS AND GROWTH:**

During the late 1990s and early 2000s, JavaScript saw significant adoption due to the "Browser Wars" between Netscape and Microsoft. JavaScript was supported by both Netscape Navigator and Microsoft's Internet Explorer, contributing to its widespread use.

**AJAX AND WEB 2.0:**

JavaScript gained renewed attention with the advent of AJAX (Asynchronous JavaScript and XML) in the mid-2000s. AJAX allowed web pages to update content asynchronously, without requiring a full page reload. This technology played a crucial role in the development of interactive and dynamic web applications, often associated with the Web 2.0 era.

**FRAMEWORKS AND LIBRARIES:**

The mid-2000s also saw the rise of JavaScript frameworks and libraries such as jQuery, Prototype, and Mootoos, which aimed to simplify web development tasks and provide cross-browser compatibility.

**NODE.JS AND SERVER-SIDE JAVASCRIPT:**

Node.js, created by Ryan Dahl and released in 2009, brought JavaScript to the server-side environment. Node.js allows developers to build scalable network applications using JavaScript on the server, further expanding the language's versatility beyond the browser.

**MODERN JAVASCRIPT AND ES6:**

In recent years, significant advancements have been made in the JavaScript language itself. ECMAScript 6 (ES6), officially known as ECMAScript 2015, introduced many new features and improvements to the language, such as arrow functions, classes, modules, and promises. Since then, new versions of ECMAScript have been released annually, bringing additional enhancements to the language.

**JAVASCRIPT EVERYWHERE:**

Today, JavaScript is ubiquitous, powering not only web applications but also mobile applications (using frameworks like React Native and Ionic), desktop applications (using frameworks like Electron), and even embedded systems. Its versatility and wide adoption make it one of the most important programming languages in modern software development.

Overall, JavaScript has undergone a remarkable evolution since its inception, shaping the way we interact with the web and powering a significant portion of the digital world.

**PRIMARY GOALS:**

JavaScript, as a programming language, has several primary goals, which have evolved over time as the language itself has developed. Some of the key primary goals of JavaScript include:

**CLIENT-SIDE INTERACTIVITY:**

One of the original goals of JavaScript was to enable interactive and dynamic behavior in web browsers. JavaScript allows developers to manipulate the content of web pages in response to user actions, such as clicks, keystrokes, and mouse movements, without needing to reload the entire page from the server. This capability enhances user experience by creating more engaging and responsive web applications.

**CROSS-PLATFORM COMPATIBILITY:**

JavaScript was designed to work across different web browsers and platforms, providing a consistent programming interface for web developers. This goal has become increasingly important as the diversity of web browsers and devices has grown. JavaScript's compatibility enables developers to write code once and deploy it across multiple platforms without significant modifications.

**VERSATILITY AND EXTENSIBILITY:**

JavaScript is a versatile language that supports various programming paradigms, including procedural, object-oriented, and functional programming. This flexibility allows developers to choose the most suitable approach for their specific needs and adapt JavaScript to different programming styles. Additionally, JavaScript's extensibility enables developers to enhance its capabilities through libraries, frameworks, and APIs, further expanding its utility.

**INTEGRATION WITH HTML AND CSS:**

JavaScript was designed to seamlessly integrate with HTML and CSS, the core technologies of the web. This integration allows developers to manipulate the structure, content, and style of web pages dynamically using JavaScript.By combining HTML for structure, CSS for presentation, and JavaScript for behavior, developers can create rich and interactive web applications.

**ASYNCHRONOUS PROGRAMMING:**

JavaScript provides support for asynchronous programming, allowing developers to execute tasks concurrently without blocking the main execution thread. This capability is essential for handling asynchronous operations such as network requests, file I/O, and user input/output. Asynchronous programming in JavaScript is typically achieved using callbacks, promises, and async/await syntax, enabling efficient and non-blocking code execution.

**SERVER-SIDE DEVELOPMENT:**

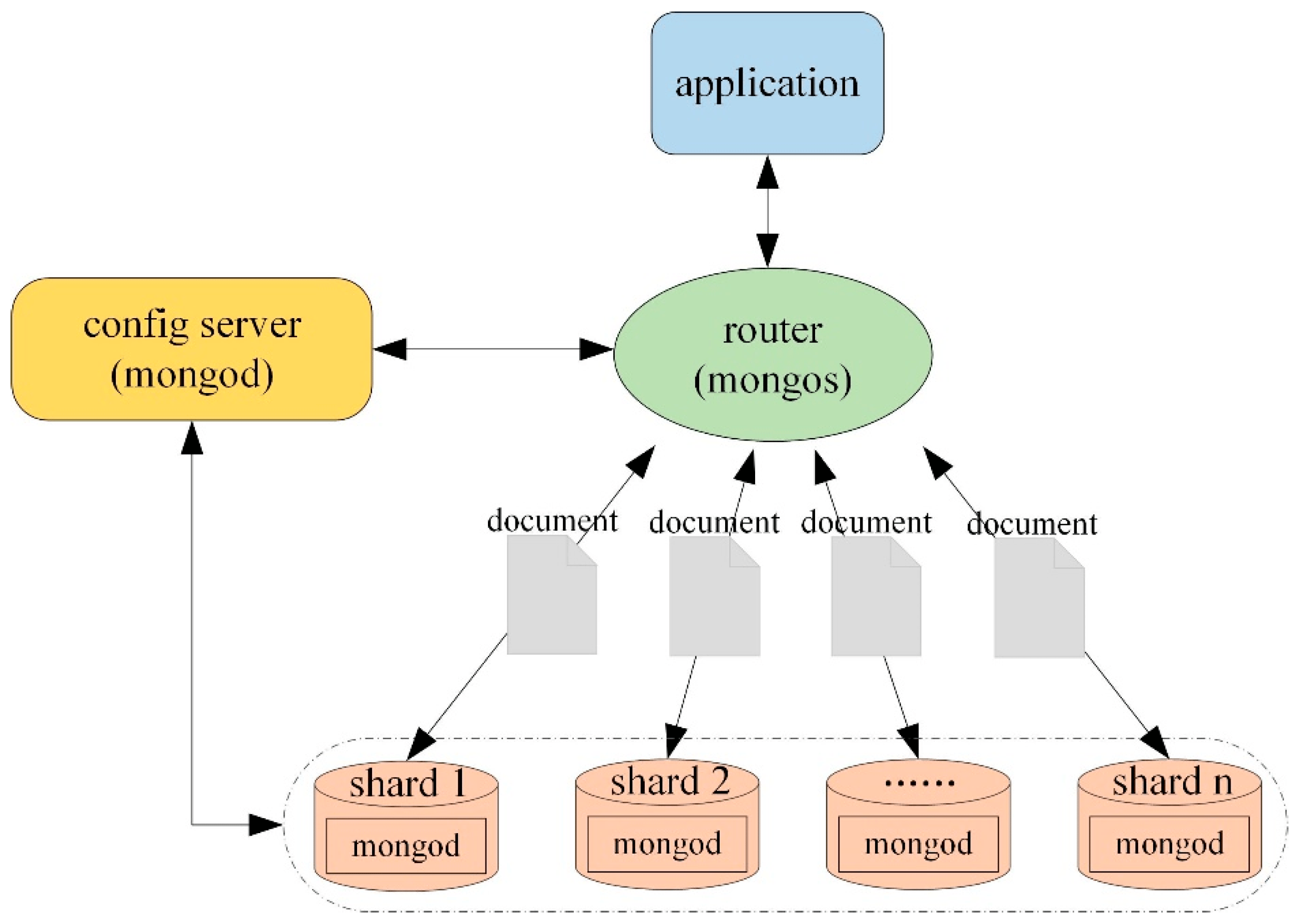
With the advent of Node.js, JavaScript has expanded its domain beyond the browser to server-side development. Node.js allows developers to build scalable and high-performance network applications using JavaScript on the server. This goal has enabled JavaScript to become a full-stack language, empowering developers to build end-to-end web applications using a single programming language.

Overall, the primary goals of JavaScript revolve around enabling interactive web experiences, fostering cross-platform compatibility, supporting various programming paradigms, integrating seamlessly with web technologies, facilitating asynchronous programming, and extending its reach to server-side development. These goals collectively contribute to JavaScript's prominence as one of the most widely used programming languages for web development

**MERN:**

The MERN stack is a popular JavaScript stack used for building web applications. It consists of four main technologies: MongoDB, Express.js, React.js, and Node.js. Each of these technologies plays a crucial role in the development process:

Together, MongoDB, Express.js, React.js, and Node.js form a powerful and modern stack for building full-stack web applications. MongoDB provides a flexible and scalable data storage solution, Express.js simplifies backend development with its minimalist framework, React.js enables the creation of dynamic and interactive user interfaces, and Node.js allows for server-side JavaScript execution and efficient handling of backend logic. The MERN stack is popular among developers for its flexibility, performance, and ease of use in building modern web applications

**MONGODB:**

**NOSQL DATABASE:**

Unlike traditional relational databases (such as MySQL or PostgreSQL), MongoDB is a NoSQL database. NoSQL databases are schema-less or schema-flexible, allowing for dynamic and unstructured data. This makes MongoDB particularly suitable for applications with evolving or unpredictable data requirements, as it can easily adapt to changes in data structure without requiring a predefined schema.

**DOCUMENT-ORIENTED:**

MongoDB stores data in a format called BSON (Binary JSON), which is a binary-encoded serialization of JSON-like documents. These documents are similar to rows in a relational database but are more flexible and can contain nested data structures, arrays, and other complex types.

Each document in MongoDB is uniquely identified by a primary key called the "\_id" field, which ensures fast and efficient retrieval of data.

**SCALABILITY:**

MongoDB is designed to scale horizontally, meaning that it can distribute data across multiple servers to handle increasing loads and storage requirements. MongoDB **uses a** technique called sharding to partition data into shards, which are distributed across multiple servers or clusters. This allows MongoDB to maintain high availability and performance even as the volume of data and the number of concurrent users grow.

**QUERYING AND INDEXING:**

MongoDB provides powerful querying capabilities, allowing developers to retrieve, filter, and manipulate data using a rich query language. MongoDB supports a wide range of query operators, aggregation pipelines, and geospatial queries for performing complex data operations. Additionally, MongoDB supports indexing, which improves query performance by creating indexes on fields or combinations of fields in the database.

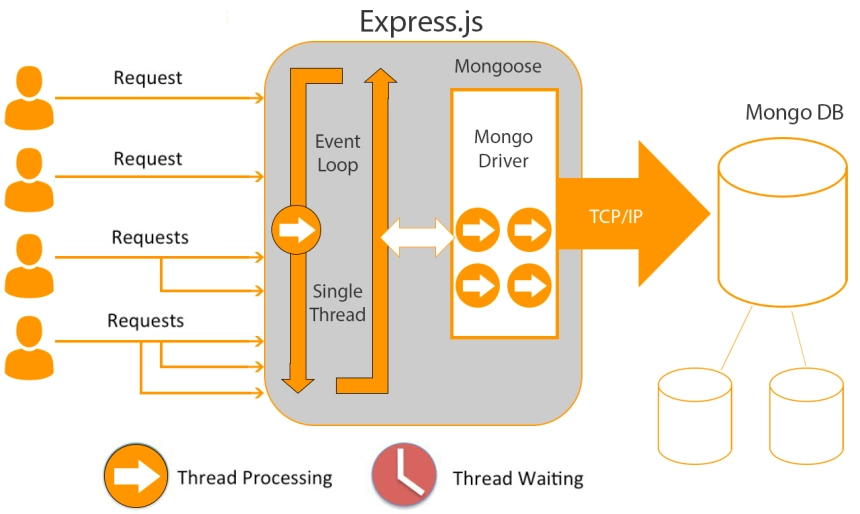
**HIGH AVAILABILITY AND FAULT TOLERANCE:**

MongoDB is designed for high availability and fault tolerance, with built-in features such as replication and automatic failover. MongoDB uses replica sets to maintain multiple copies of data across different servers or nodes. In the event of a server failure, MongoDB automatically promotes a secondary node to primary status, ensuring continuous availability and data consistency.

**COMMUNITY AND ECOSYSTEM:**

MongoDB has a vibrant and active community of developers, users, and contributors. The MongoDB community provides extensive documentation, tutorials, and resources to help developers get started with MongoDB and troubleshoot any issues they encounter. Additionally, MongoDB offers a rich ecosystem of tools, drivers, and integrations for various programming languages and frameworks, making it easy to use MongoDB with popular development stacks and platforms.

Overall, MongoDB is a powerful, flexible, and scalable NoSQL database solution suitable for a wide range of applications, from small-scale projects to enterprise-level deployments. Its document-oriented data model, rich querying capabilities, scalability features, and strong community support make it a popular choice among developers for building modern web applications, mobile apps, and other data-intensive systems

**EXPRESS:**

Express.js, commonly referred to as Express, is a minimalist web application framework for Node.js. It provides a robust set of features for building web servers and APIs with Node.js, simplifying the process of handling HTTP requests, defining routes, and integrating middleware.

Here's a breakdown of some key features and concepts of Express:

**ROUTING:**

Express allows developers to define routes to handle different HTTP requests (GET, POST, PUT, DELETE, etc.) to specific URL paths.

Route handlers can be defined to execute code logic when a particular route is accessed. Express provides a clean and intuitive syntax for defining routes, making it easy to organize and manage the application's endpoints.

**MIDDLEWARE:**

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. Middleware functions can perform tasks such as logging, authentication, data parsing, error handling, and more. Express allows developers to define middleware functions and apply them to specific routes or to the entire application's request pipeline, providing flexibility and modularity in building web servers.

**HTTP SERVER:**

Express simplifies the process of creating an HTTP server in Node.js. With just a few lines of code, developers can create a new Express application, define routes, and start the server to listen for incoming HTTP requests. Express abstracts away many of the low-level details of working with HTTP in Node.js, allowing developers to focus on building the application's business logic rather than dealing with network programming complexities.

**TEMPLATE ENGINES:**

While Express itself does not include a built-in template engine, it provides support for integrating popular template engines such as Pug, EJS, Handlebars, and others. Template engines allow developers to generate dynamic HTML content on the server and render it to the client in response to HTTP requests. Express makes it easy to configure and use template engines, enabling the creation of dynamic web pages and views.

**ERROR HANDLING:**

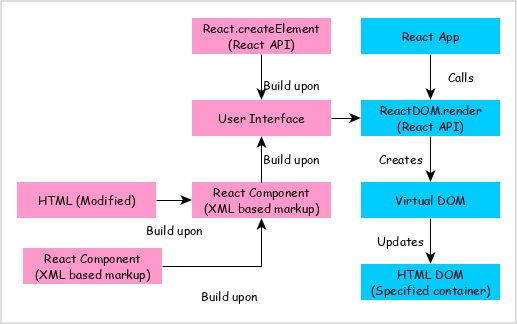
Express includes built-in mechanisms for handling errors that occur during request processing. Developers can define error-handling middleware functions to catch errors and respond to clients with appropriate error messages or status codes. Express also provides a default error-handling middleware that can be customized to suit the application's error-handling needs.

**STATIC FILE SERVING:**

Express allows developers to serve static files (such as HTML, CSS, JavaScript, images, and other assets) from a directory on the server. This is useful for serving client-side resources and building single-page applications (SPAs) with front-end frameworks like React.js or Angular. Express provides a built-in middleware function (express.static) for serving static files with minimal configuration.

Overall, Express.js is a powerful and flexible web application framework for Node.js, providing developers with a simple and intuitive way to build web servers, APIs, and full-stack web applications. Its minimalist design, robust feature set, and strong community support make it a popular choice among developers for building modern web applications and APIs.

**REACT:**



React.js, commonly referred to as React, is a JavaScript library for building user interfaces. Developed by Facebook, React has gained widespread popularity among developers due to its efficient and declarative approach to building interactive and dynamic UIs.

Here's a breakdown of some key concepts and features of React:

**COMPONENT-BASED ARCHITECTURE:**

React follows a component-based architecture, where UIs are built using reusable components. Components are self-contained units of UI that encapsulate their own logic and rendering behavior. This promotes code reusability, modularity, and maintainability, as developers can create complex UIs by composing smaller, reusable components together.

**VIRTUAL DOM (DOCUMENT OBJECT MODEL):**

One of React's core innovations is its use of a virtual DOM. The virtual DOM is a lightweight, in-memory representation of the actual DOM. When changes are made to the UI, React updates the virtual DOM rather than directly manipulating the browser's DOM. React then compares the virtual DOM with the previous version and calculates the most efficient way to update the actual DOM. This approach minimizes DOM manipulation and leads to better performance, especially in applications with dynamic and frequently changing UIs.

**DECLARATIVE SYNTAX:**

React promotes a declarative programming paradigm, where developers describe the desired UI state and React takes care of updating the DOM to match that state. This is in contrast to imperative programming, where developers specify each step of the UI update process. With React's declarative syntax, developers can focus on what the UI should look like based on the application's state, rather than worrying about how to achieve that state. This makes the code more predictable, easier to understand, and less error-prone.

**COMPONENT LIFECYCLE METHODS:**

React components have lifecycle methods that allow developers to hook into different stages of a component's lifecycle, such as when it is mounted, updated, or unmounted. Lifecycle methods provide opportunities to perform tasks such as fetching data, updating the UI in response to changes, or cleaning up resources when a component is no longer needed. This allows developers to create components that are responsive, interactive, and well-behaved throughout their lifecycle.

**STATE MANAGEMENT:**

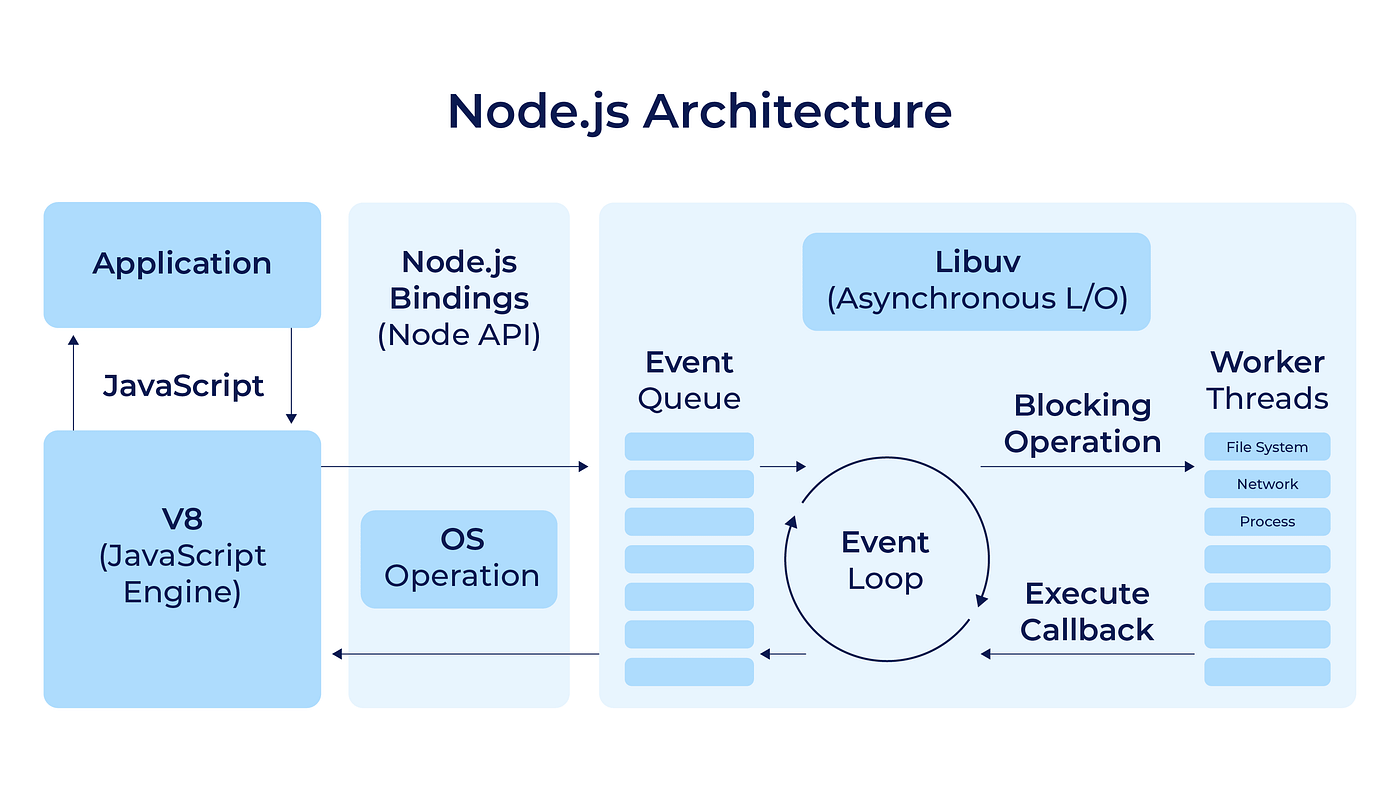
React provides a built-in mechanism for managing component state. State represents the data that a component needs to render its UI and respond to user interactions. By using React's state management features, developers can update and synchronize the state of their components in response to user input, API calls, or other events. React's unidirectional data flow ensures that changes to component state propagate downwards through the component hierarchy, making it easier to reason about the application's state.

**JSX (JAVASCRIPT XML):**

React uses JSX, an extension of JavaScript syntax that allows developers to write HTML-like code directly within JavaScript. JSX makes it easy to define UI components and their structure, while still leveraging the full power of JavaScript for logic and data manipulation. JSX code is transpiled into regular JavaScript code by tools like Babel before being executed by the browser.

Overall, React.js is a powerful and efficient library for building user interfaces in JavaScript applications. Its component-based architecture, virtual DOM, declarative syntax, and rich ecosystem of tools and libraries make it a popular choice among developers for building modern web applications, single-page applications (SPAs), and mobile apps.

NODE:



Node.js is a server-side JavaScript runtime environment that allows developers to run JavaScript code outside of a web browser. Here's a comprehensive explanation of Node.js:

**JAVASCRIPT RUNTIME:**

Node.js is built on Chrome's V8 JavaScript engine, which is known for its speed and efficiency in executing JavaScript code. Node.js allows developers to use JavaScript for server-side programming, enabling full-stack JavaScript development. This means that developers can use the same language (JavaScript) for both client-side and server-side code, streamlining the development process and promoting code reuse.

**EVENT-DRIVEN ARCHITECTURE:**

Node.js follows an event-driven, non-blocking I/O model, which makes it highly scalable and efficient for handling concurrent operations. In Node.js, asynchronous programming is fundamental, meaning that I/O operations such as reading from a file or making a network request do not block the execution of other code. Instead, callbacks or Promises are used to handle the completion of asynchronous tasks. This allows Node.js to handle a large number of connections simultaneously without getting bogged down by blocking operations.

**MODULES AND NPM:**

Node.js has a modular architecture, where functionality is organized into small, reusable modules. These modules can be easily imported into Node.js applications using the require keyword. Additionally, Node.js comes with a built-in package manager called npm (Node Package Manager), which allows developers to install, manage, and share third-party modules and libraries. npm provides access to a vast ecosystem of open-source packages, making it easy for developers to find solutions to common problems and accelerate the development process.

**SERVER-SIDE DEVELOPMENT:**

Node.js is commonly used for building web servers and back-end APIs. With its event-driven architecture and non-blocking I/O, Node.js excels at handling I/O-intensive tasks such as serving HTTP requests, handling database operations, and processing file uploads. Developers can use frameworks like Express.js to build robust and scalable web applications with Node.js. Express.js provides a minimalist and flexible framework for building web servers and defining routes, middleware, and handlers.

**REAL-TIME APPLICATIONS:**

Node.js is well-suited for building real-time applications such as chat applications, online gaming platforms, and collaborative editing tools. Its event-driven architecture and support for WebSocket’s enable developers to build applications that require bi-directional communication between the client and server in real-time. Libraries like Socket.IO simplify the process of implementing real-time features in Node.js applications by providing abstractions for handling WebSocket connections and managing communication between clients and servers.

**COMMAND-LINE TOOLS:**

Node.js can also be used to build command-line tools and scripts for automating tasks, data processing, and system administration.

The built-in fs (file system) module provides APIs for interacting with the file system, making it easy to read from and write to files. Additionally, Node.js can be used with third-party libraries like Commander.js and yargs to create interactive command-line interfaces with options, arguments, and help documentation.

Overall, Node.js is a versatile and powerful platform for building scalable, high-performance web applications, APIs, and command-line tools using JavaScript. Its event-driven architecture, non-blocking I/O, modular design, and vibrant ecosystem of modules and libraries make it a popular choice among developers for a wide range of applications.

**MERN FLOW DIAGRAM:**

A diagram of a structure

Description automatically generated

# **CHAPTER 4 MPDULES DESCRIPTION**

This Project contains the following modules.

**FARMER:**

* User Authentication and Profile Management
* Product Listings and Management
* Notification

**USER:**

* User Authentication and Profile Management
* Local Product Search and Filtering
* Notification

**FARMER USER AUTHENTICATION AND PROFILE MANAGEMENT:**

**USER REGISTRATION:**

This feature allows new users to create accounts within the application. Users typically provide information such as their name, email address, username, and password during the registration process. The application may also include additional fields for capturing other relevant details, such as location, preferences, or user type (e.g., farmer or consumer). After successful registration, the user's information is stored securely in the database.

**LOGIN:**

Once registered, users can log in to the application using their credentials (e.g., username/email and password). The login process verifies the user's identity by checking their credentials against the stored data in the database. Upon successful authentication, the user is granted access to their account and any associated features or content. To ensure security, login systems often implement measures such as password hashing and salting to protect user credentials from unauthorized access.

**PROFILE MANAGEMENT:**

Users should have the ability to manage their profiles within the application. This includes updating personal information such as contact details, preferences, profile pictures, and any other relevant data. Users may also have the option to customize their profile settings, such as notification preferences or privacy settings. Profile management functionality empowers users to maintain accurate and up-to-date information within the application.

**ROLE-BASED ACCESS CONTROL (OPTIONAL):**

Depending on the application's requirements, role-based access control (RBAC) may be implemented to restrict access to certain features or content based on user roles or permissions. For example, in a platform where farmers and consumers have different roles, RBAC could be used to grant specific privileges to each user type. This helps ensure that users only have access to the functionalities relevant to their role within the application.

Overall, User Authentication and Profile Management form the foundation of user interaction and personalization in web applications. By providing secure registration, login, and profile management functionalities, applications can offer a seamless and personalized experience to their users while prioritizing the security and privacy of their data.

**FARMER PRODUCT LISTINGS AND MANAGEMENT:**

The "Farmer Product Listings and Management" module is an essential component of agricultural e-commerce platforms or marketplaces that connect farmers directly with consumers. Here's a more detailed explanation of its functionalities:

**PRODUCT LISTING:**

Farmers can create listings for the agricultural products they have available for sale. This typically involves providing details such as the product name, description, quantity available, pricing information (e.g., unit price or total price), and any other relevant **attributes (e.g.,** product category, quality, origin, etc.). The listings may also include images to showcase the products to potential buyers.

**PRODUCT MANAGEMENT:**

Farmers have the ability to manage their product listings within the application. This includes actions such as adding new products, updating existing listings with revised information (e.g., changes in quantity or price), and removing products that are no longer available for sale. Product management functionalities empower farmers to keep their listings accurate and up-to-date, ensuring a positive experience for buyers.

**INVENTORY TRACKING:**

The module may include inventory management features to help farmers track the availability of their products. This involves monitoring the quantity of each product listed for sale and updating the inventory levels as products are sold or restocked. Inventory tracking ensures that farmers can accurately reflect product availability in their listings and prevent overselling or stockouts.

**ORDER MANAGEMENT:**

When a consumer expresses interest in purchasing a product listed by a farmer, the farmer receives an order notification. The farmer can then review the order details, including the product(s) requested, quantity, and buyer information. Based on this information, the farmer can choose to accept or decline the order. Accepting the order confirms the sale, while declining the order indicates that the product is not available or the farmer is unable to fulfill the request.

**COMMUNICATION:**

The module may facilitate communication between farmers and buyers to discuss product details, negotiate prices, arrange delivery or pickup options, and address any questions or concerns. Communication channels such as messaging or chat functionalities enable real-time interaction between parties, fostering trust and transparency in the buying process.

**TRANSACTION HISTORY:**

The application may maintain a transaction history for farmers, providing a record of past sales and purchases. This allows farmers to track their sales performance, monitor revenue, and analyze trends over time. Transaction history also serves as a valuable reference for financial reporting and accounting purposes.

Overall, the "Farmer Product Listings and Management" module streamlines the process of selling agricultural products directly from farmers to consumers. By providing farmers with tools to list, manage, and sell their products effectively, the module contributes to the efficiency, transparency, and profitability of agricultural e-commerce platforms. Additionally, it enhances the accessibility of fresh, locally sourced produce for consumers while supporting farmers in reaching broader markets and increasing their revenue opportunities.

**FARMER NOTIFICATION:**

The "Farmer Notification" module is designed to provide farmers with timely notifications about new orders placed by consumers, as well as important updates related to their products and transactions. Here's a more detailed explanation of its functionalities:

**ORDER NOTIFICATIONS:**

When a consumer places an order for a product listed by the farmer, the farmer receives a notification alerting them of the new order. This notification typically includes details such as the product(s) ordered, quantity, buyer information (e.g., username, contact details), and any special instructions or preferences provided by the buyer.

Order notifications ensure that farmers are promptly informed of incoming orders and can take appropriate action to fulfill them.

**PRODUCT UPDATES:**

In addition to order notifications, the module may also send notifications to farmers regarding updates or changes to their product listings. For example, if a product's quantity or price is modified by the farmer, they may receive a notification confirming the update. Similarly, if a product is out of stock or no longer available for sale, the farmer would receive a notification informing them of the status change. Product updates help farmers stay informed about the availability and status of their products in real-time.

**ACCEPT OR DECLINE ORDERS:**

Upon receiving an order notification, the farmer has the capability to review the order details and decide whether to accept or decline the order. If the farmer chooses to accept the order, it confirms the sale, and the buyer is notified that their order has been accepted. Conversely, if the farmer declines the order (e.g., due to product unavailability or other reasons), the buyer is notified that their order cannot be fulfilled. Accepting or declining orders gives farmers control over their sales process and allows them to manage their inventory effectively.

**CUSTOMIZATION AND PREFERENCES:**

The module may offer options for farmers to customize their notification preferences based on their preferences and needs. Farmers can choose to receive notifications via email, SMS, push notifications through a mobile app, or any other preferred communication channel. Additionally, farmers may have the ability to configure notification settings, such as frequency, urgency, and types of notifications they wish to receive (e.g., order notifications, product updates, etc.).

**REAL-TIME COMMUNICATION:**

In some cases, the notification module may integrate with communication tools or messaging features to facilitate real-time communication between farmers and buyers. For example, if a buyer has questions or requests related to their order, they may be able to send a message directly to the farmer through the notification interface. This enables efficient communication and collaboration between parties, leading to smoother transactions and enhanced customer satisfaction.

Overall, the "Farmer Notification" module plays a crucial role in keeping farmers informed about important events and activities related to their products and orders. By providing timely notifications and communication channels, the module helps farmers manage their sales effectively, respond promptly to customer inquiries, and maintain a positive user experience for buyers

**USER LOCAL PRODUCT SEARCH AND FILTERING:**

The "Local Product Search and Filtering" feature enhances the user experience by enabling consumers to search for agricultural products based on their location or specified areas, thereby facilitating the purchase of locally produced goods. Here's a more detailed explanation of its functionalities:

**LOCATION-BASED SEARCH:**

Users, primarily consumers, can initiate searches for agricultural products based on their current location or specified areas. This feature leverages location-based services to identify the user's geographical coo rdinates or allows users to input their desired location manually. By searching for products within their local area, consumers can find nearby farms or sellers offering fresh produce, promoting the consumption of locally sourced goods and supporting local farmers.

**FILTERING OPTIONS:**

The search feature includes filtering options to refine search results based on various criteria. Users can filter products based on product types (e.g., fruits, vegetables, dairy), project categories (e.g., organic, sustainable farming), or other relevant attributes such as price range, availability, or seller ratings. These filtering options help users narrow down their search results to find products that meet their specific preferences and requirements.

**ADVANCED SEARCH CAPABILITIES:**

In addition to basic search and filtering options, the feature may include advanced search capabilities to provide users with more precise results. Advanced search functionalities may include keyword search, allowing users to enter specific product names or descriptions to find relevant items quickly. Users may also have the option to save their search criteria for future reference or set up search alerts to receive notifications when new products matching their criteria are listed.

**PRODUCT DETAILS AND LISTINGS:**

Upon initiating a search, users are presented with a list of matching product listings, each displaying relevant details such as product name, description, price, seller information, and location. Users can click on individual listings to view more detailed information about the product, including images, specifications, and seller reviews. This allows users to make informed purchasing decisions based on the available information.

**BUYING OPTIONS:**

Once users find a product they are interested in purchasing, the feature facilitates the buying process by providing convenient options to add the product to their cart or initiate the purchase directly. Users can select the desired quantity and any specific preferences (e.g., size, color) before proceeding to checkout. Integration with payment gateways allows users to complete the purchase securely, making it easy and hassle-free to buy agricultural products online.

**INTEGRATION WITH MAPPING SERVICES:**

To enhance the user experience further, the feature may integrate with mapping services such as Google Maps or geolocation APIs to display search results visually on a map interface. Users can see the locations of nearby farms or sellers offering the desired products, making it easier to identify convenient pickup points or delivery options based on their location.

Overall, the "Local Product Search and Filtering" feature empowers consumers to discover and purchase agricultural products conveniently while supporting local farmers and promoting sustainable consumption practices. By combining location-based search, advanced filtering options, and seamless buying experiences, the feature enhances user engagement and satisfaction in agricultural e-commerce platforms.

**USER NOTIFICATION:**

The "Notification" feature provides users with the capability to review various types of notifications related to orders, users, and products within the application. Here's a more detailed explanation of its functionalities:

**ORDER NOTIFICATIONS:**

Users receive notifications regarding their orders, including updates on order status, confirmation of successful purchases, and any changes or actions required from the user's end. For example, users may receive notifications when their order is confirmed, shipped, out for delivery, or delivered. These notifications keep users informed about the progress of their orders and provide timely updates throughout the order fulfillment process.

**PRODUCT NOTIFICATIONS:**

In addition to order and user notifications, users may receive notifications related to products they are interested in or have interacted with. For example, users may receive notifications when a product they viewed or added to their wishlist is back in stock, has a price drop, or has new reviews or ratings. These notifications help users stay updated on products of interest and make informed purchasing decisions.

**NOTIFICATION MANAGEMENT:**

Users have the capability to manage their notification preferences and settings within the application. This includes options to customize which types of notifications they wish to receive, the frequency of notifications, and the preferred communication channels.

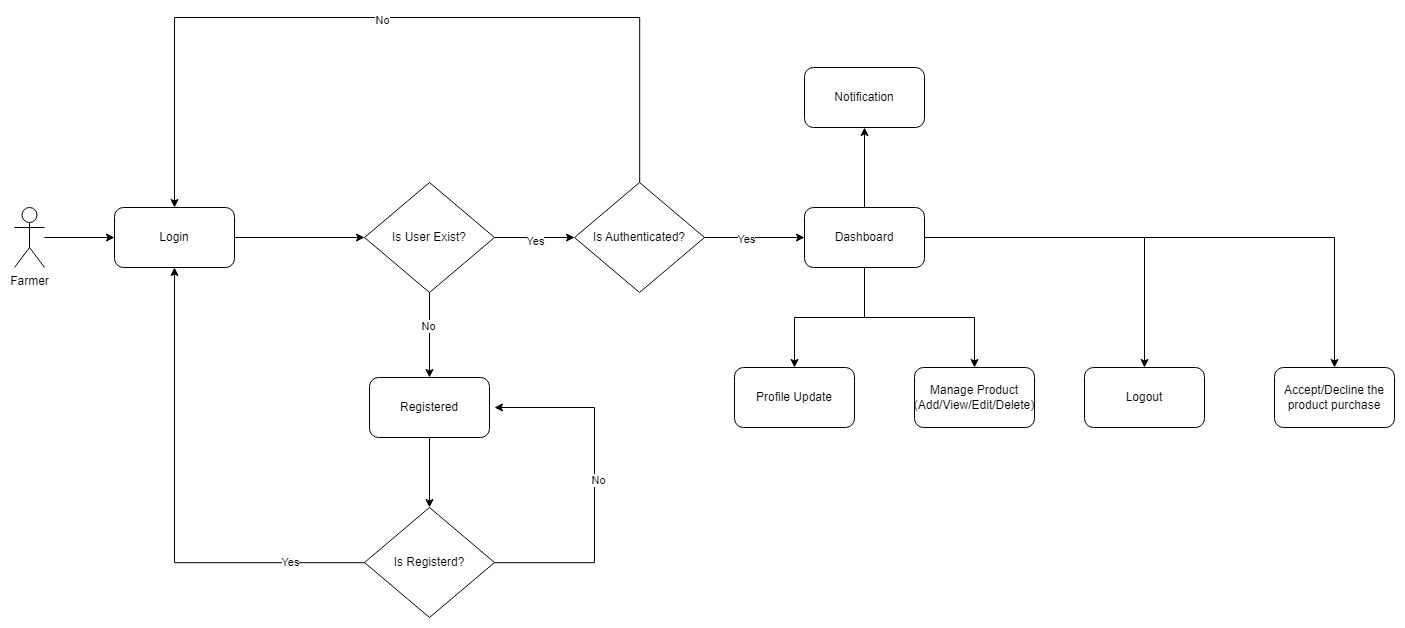
# **CHAPTER 5 DATA FLOW DIAGRAM**

**USER FLOW DIAGRAM:**

****

A user flow diagram, also known as a user journey or user flowchart, is a visual representation of the steps a user takes to complete a task or achieve a goal within a system or application. It maps out the sequence of actions and decisions that a user makes as they interact with the interface, guiding them from the initial entry point to the final outcome

**FAMER FLOW DIAGRAM:**

****

A farmer flow diagram outlines the various steps and interactions involved in the user journey specifically tailored for farmers within an application or platform. Here's an explanation of a typical farmer flow diagramterface, guiding them from the initial entry point to the final outcome

# **CHAPTER 6 IMPLEMENTATION**

Guidelines that drive the implementation and analysis include:

* Any difficulty in design, coding and testing a modification should signal the need for redesign or re-coding.
* Modifications should fit easily into isolated and easy-to-find modules. If they do not, some redesign is needed.
* Modifications to tables should be especially easy to make. If any table modification is not quickly and easily done, redesign is indicated.
* Modifications should become easier to make as the iterations progress. If they are not, there is a basic problem such as a design flaw or a proliferation of patches.
* Patches should normally be allowed to exist for only one or two iterations. Patches may be necessary to avoid redesigning during an implementation phase.
* The existing implementation should be analysed frequently to determine how well it measures up to project goals.
* Program analysis facilities should be used whenever available to aid in the analysis of partial implementations.

# **CHAPTER 7 SOURCE CODE**

**UI FRONT-END:**

**SIN-UP:**

**/\* eslint-disable import/no-unresolved \*/**

**// eslint-disable-next-line**

**import { postData } from 'src/services/apiService';**

**import { useState } from 'react';**

**import Box from '@mui/material/Box';**

**import { Tooltip } from '@mui/material';**

**import Card from '@mui/material/Card';**

**import Stack from '@mui/material/Stack';**

**import Button from '@mui/material/Button';**

**import Divider from '@mui/material/Divider';**

**import TextField from '@mui/material/TextField';**

**import Typography from '@mui/material/Typography';**

**import IconButton from '@mui/material/IconButton';**

**import LoadingButton from '@mui/lab/LoadingButton';**

**import { alpha, useTheme } from '@mui/material/styles';**

**import InputAdornment from '@mui/material/InputAdornment';**

**import { useRouter } from 'src/routes/hooks';**

**import { bgGradient } from 'src/theme/css';**

**import Logo from 'src/components/logo';**

**import Iconify from 'src/components/iconify';**

**import { toast } from 'react-toastify';**

**import 'react-toastify/dist/ReactToastify.css';**

**// ----------------------------------------------------------------------**

**export default function SignUpView() {**

**const theme = useTheme();**

**const router = useRouter();**

**const [formData, setFormData] = useState({ name: '', phoneNumber: '', email: '', password: '', repassword: '', role: '' });**

**const [formErrors, setFormErrors] = useState({**

**name: '',**

**email: '',**

**password: '',**

**phoneNumber: '',**

**repassword: '',**

**role: ''**

**});**

**const handleChange = (event) => {**

**const { name, value } = event.target;**

**setFormData({ ...formData, [name]: value }); // Update the respective field in the formData state**

**};**

**const [showPassword, setShowPassword] = useState(false);**

**const [showRePassword, setShowRePassword] = useState(false);**

**const handleClick = async () => {**

**const isValid = validateForm();**

**console.log("FormErrors", formErrors)**

**if (isValid) {**

**try {**

**const response = await postData('register', formData);**

**console.log('api response', response);**

**router.push('/login');**

**} catch (error) {**

**console.error('Error fetching data:', error);**

**}**

**} else {**

**toast.error('Please fill in all required fields correctly');**

**}**

**};**

**const onRoleSelect = async (value) => {**

**setFormData({ ...formData, "role": value });**

**}**

**const validateForm = () => {**

**let valid = true;**

**const errors = {};**

**if (!formData.email) {**

**errors.email = 'Email is required';**

**valid = false;**

**}**

**if (!formData.name) {**

**errors.name = 'Name is required';**

**valid = false;**

**}**

**if (!formData.phoneNumber) {**

**errors.phoneNumber = 'Phone number is required';**

**valid = false;**

**}**

**if (!formData.password) {**

**errors.password = 'Password is required';**

**valid = false;**

**}**

**if (!formData.repassword) {**

**errors.repassword = 'Re-Password is required';**

**valid = false;**

**}**

**console.log("formData", formData)**

**if (!formData.role) {**

**errors.role = "Please select role";**

**valid = false;**

**}**

**// Add more validation rules for other fields as needed...**

**setFormErrors(errors);**

**return valid;**

**};**

**const renderForm = (**

**<>**

**<Stack spacing={3}>**

**<TextField name="name" label="Name" value={formData.name}**

**error={!!formErrors.name}**

**helperText={formErrors.name}**

**onChange={handleChange} />**

**<TextField**

**name="email"**

**label="Email address"**

**value={formData.email}**

**error={!!formErrors.email}**

**helperText={formErrors.email}**

**onChange={handleChange}**

**/>**

**<TextField**

**name="phoneNumber"**

**label="Phone Number"**

**value={formData.phoneNumber}**

**error={!!formErrors.phoneNumber}**

**helperText={formErrors.phoneNumber}**

**onChange={handleChange}**

**/>**

**<TextField**

**name="password"**

**label="Password"**

**type={showPassword ? 'text' : 'password'}**

**value={formData.password}**

**error={!!formErrors.password}**

**helperText={formErrors.password}**

**onChange={handleChange}**

**InputProps={{**

**endAdornment: (**

**<InputAdornment position="end">**

**<IconButton onClick={() => setShowPassword(!showPassword)} edge="end">**

**<Iconify icon={showPassword ? 'eva:eye-fill' : 'eva:eye-off-fill'} />**

**</IconButton>**

**</InputAdornment>**

**),**

**}}**

**/>**

**<TextField**

**name="repassword"**

**label="Re-Password"**

**type={showRePassword ? 'text' : 'password'}**

**value={formData.repassword}**

**error={!!formErrors.repassword}**

**helperText={formErrors.repassword}**

**onChange={handleChange}**

**InputProps={{**

**endAdornment: (**

**<InputAdornment position="end">**

**<IconButton onClick={() => setShowRePassword(!showRePassword)} edge="end">**

**<Iconify icon={showRePassword ? 'eva:eye-fill' : 'eva:eye-off-fill'} />**

**</IconButton>**

**</InputAdornment>**

**),**

**}}**

**/>**

**</Stack>**

**<Stack direction="row" alignItems="center" justifyContent="flex-end" sx={{ my: 3 }} />**

**<LoadingButton**

**fullWidth**

**size="large"**

**type="submit"**

**variant="contained"**

**color="inherit"**

**onClick={handleClick}**

**>**

**Register**

**</LoadingButton>**

**</>**

**);**

**/\* name,number,email,password, re-password \*/**

**return (**

**<Box**

**sx={{**

**...bgGradient({**

**color: alpha(theme.palette.background.default, 0.9),**

**imgUrl: '/assets/background/overlay\_4.jpg',**

**}),**

**minHeight: '100%',**

**overflowY: 'auto',**

**}}**

**>**

**<Logo**

**sx={{**

**position: 'fixed',**

**top: { xs: 16, md: 24 },**

**left: { xs: 16, md: 24 },**

**}}**

**/>**

**<Stack alignItems="center" justifyContent="center" sx={{ height: 1 }}>**

**<Card**

**sx={{**

**p: 5,**

**width: 1,**

**maxWidth: 420,**

**}}**

**>**

**<Typography variant="h4">Register for Farmers Market</Typography>**

**<Typography variant="body2" sx={{ mt: 2, mb: 5 }}>**

**Already have an account?**

**<a href="/login">Login here</a>**

**</Typography>**

**<Stack direction="row" spacing={2}>**

**<Tooltip title="User">**

**<Button**

**fullWidth**

**size="large"**

**color="inherit"**

**variant="outlined"**

**sx={{ borderColor: formData.role === 2 ? theme.palette.primary.dark : alpha(theme.palette.grey[500], 0.16), }}**

**onClick={() => onRoleSelect(2)}**

**>**

**<Iconify icon="fluent-emoji:person-light" color="#DF3E30" />**

**</Button>**

**</Tooltip>**

**<Tooltip title="Farmer">**

**<Button**

**fullWidth**

**size="large"**

**color="inherit"**

**variant="outlined"**

**sx={{ borderColor: formData.role === 1 ? theme.palette.primary.dark : alpha(theme.palette.grey[500], 0.16), }}**

**onClick={() => onRoleSelect(1)}**

**>**

**<Iconify icon="twemoji:farmer" color="#1877F2" />**

**</Button>**

**</Tooltip>**

**</Stack>**

**{formErrors.role !== '' && <div className="error">{formErrors.role}</div>}**

**<Divider sx={{ my: 3 }} />**

**{renderForm}**

**</Card>**

**</Stack>**

**</Box>**

**);}**

**LOG-IN:**

**// eslint-disable-next-line**

**import { postData } from 'src/services/apiService';**

**import { useState } from 'react';**

**import Box from '@mui/material/Box';**

**import Card from '@mui/material/Card';**

**import Stack from '@mui/material/Stack';**

**import TextField from '@mui/material/TextField';**

**import Typography from '@mui/material/Typography';**

**import IconButton from '@mui/material/IconButton';**

**import LoadingButton from '@mui/lab/LoadingButton';**

**import { alpha, useTheme } from '@mui/material/styles';**

**import InputAdornment from '@mui/material/InputAdornment';**

**import { useRouter } from 'src/routes/hooks';**

**import { bgGradient } from 'src/theme/css';**

**import Logo from 'src/components/logo';**

**import Iconify from 'src/components/iconify';**

**import { toast } from 'react-toastify';**

**import 'react-toastify/dist/ReactToastify.css';**

**// ----------------------------------------------------------------------**

**export default function LoginView() {**

**const theme = useTheme();**

**const router = useRouter();**

**const [formData, setFormData] = useState({ email: '', password: '' });**

**const [formErrors, setFormErrors] = useState({**

**email: '',**

**password: '',**

**});**

**const handleChange = (event) => {**

**const { name, value } = event.target;**

**setFormData({ ...formData, [name]: value }); // Update the respective field in the formData state**

**};**

**const [showPassword, setShowPassword] = useState(false);**

**const handleClick = async () => {**

**const isValid = validateForm();**

**if (isValid) {**

**try {**

**const response = await postData('/login', formData);**

**localStorage.setItem('loginDetails', JSON.stringify(response));**

**console.log('api response', response);**

**router.push('/dashboard');**

**} catch (error) {**

**console.error('Error fetching data:', error);**

**}**

**} else {**

**toast.error('Please fill in all required fields correctly');**

**}**

**};**

**const validateForm = () => {**

**let valid = true;**

**const errors = {};**

**if (!formData.email) {**

**errors.email = 'Email is required';**

**valid = false;**

**}**

**if (!formData.password) {**

**errors.password = 'Password is required';**

**valid = false;**

**}**

**// Add more validation rules for other fields as needed...**

**setFormErrors(errors);**

**return valid;**

**};**

**/\* email ,password \*/**

**/\* profile - photo, name, dob , age, gender, address, pincode , submit btn \*/**

**const renderForm = (**

**<>**

**<Stack spacing={3}>**

**<TextField**

**name="email"**

**label="Email address"**

**error={!!formErrors.email}**

**helperText={formErrors.email}**

**value={formData.email}**

**onChange={handleChange}**

**/>**

**<TextField**

**name="password"**

**label="Password"**

**type={showPassword ? 'text' : 'password'}**

**value={formData.password}**

**error={!!formErrors.password}**

**helperText={formErrors.password}**

**onChange={handleChange}**

**InputProps={{**

**endAdornment: (**

**<InputAdornment position="end">**

**<IconButton onClick={() => setShowPassword(!showPassword)} edge="end">**

**<Iconify icon={showPassword ? 'eva:eye-fill' : 'eva:eye-off-fill'} />**

**</IconButton>**

**</InputAdornment>**

**),**

**}}**

**/>**

**</Stack>**

**<Stack direction="row" alignItems="center" justifyContent="flex-end" sx={{ my: 3 }} />**

**<LoadingButton**

**fullWidth**

**size="large"**

**type="submit"**

**variant="contained"**

**color="inherit"**

**onClick={handleClick}**

**>**

**Login**

**</LoadingButton>**

**</>**

**);**

**return (**

**<Box**

**sx={{**

**...bgGradient({**

**color: alpha(theme.palette.background.default, 0.9),**

**imgUrl: '/assets/background/overlay\_4.jpg',**

**}),**

**height: 1,**

**}}**

**>**

**<Logo**

**sx={{**

**position: 'fixed',**

**top: { xs: 16, md: 24 },**

**left: { xs: 16, md: 24 },**

**}}**

**/>**

**<Stack alignItems="center" justifyContent="center" sx={{ height: 1 }}>**

**<Card**

**sx={{**

**p: 5,**

**width: 1,**

**maxWidth: 420,**

**}}**

**>**

**<Typography variant="h4">Sign in to Farmers Market</Typography>**

**<Typography variant="body2" sx={{ mt: 2, mb: 5 }}>**

**Don’t have an account?**

**<a href="/signup">Get started</a>**

**</Typography>**

**{renderForm}**

**</Card>**

**</Stack>**

**</Box>**

**);**

**}**

**USERS:**

**import Card from '@mui/material/Card';**

**import Stack from '@mui/material/Stack';**

**import Button from '@mui/material/Button';**

**import Container from '@mui/material/Container';**

**import Typography from '@mui/material/Typography';**

**import Iconify from 'src/components/iconify';**

**// ----------------------------------------------------------------------**

**export default function UserPage() {**

**return (**

**<Container>**

**<Stack direction="row" alignItems="center" justifyContent="space-between" mb={5}>**

**<Typography variant="h4">Users</Typography>**

**<Button variant="contained" color="inherit" startIcon={<Iconify icon="eva:plus-fill" />}>**

**New User**

**</Button>**

**</Stack>**

**<Card/>**

**</Container>**

**);**

**}**

**import PropTypes from 'prop-types';**

**import TableRow from '@mui/material/TableRow';**

**import TableCell from '@mui/material/TableCell';**

**// ----------------------------------------------------------------------**

**export default function TableEmptyRows({ emptyRows, height }) {**

**if (!emptyRows) {**

**return null;**

**}**

**return (**

**<TableRow**

**sx={{**

**...(height && {**

**height: height \* emptyRows,**

**}),**

**}}**

**>**

**<TableCell colSpan={9} />**

**</TableRow>**

**);**

**}**

**TableEmptyRows.propTypes = {**

**emptyRows: PropTypes.number,**

**height: PropTypes.number,**

**};**

import PropTypes from 'prop-types';

import Paper from '@mui/material/Paper';

import TableRow from '@mui/material/TableRow';

import TableCell from '@mui/material/TableCell';

import Typography from '@mui/material/Typography';

// ----------------------------------------------------------------------

export default function TableNoData({ query }) {

return (

<TableRow>

<TableCell align="center" colSpan={6} sx={{ py: 3 }}>

<Paper

sx={{

textAlign: 'center',

}}

>

<Typography variant="h6" paragraph>

Not found

</Typography>

<Typography variant="body2">

No results found for &nbsp;

<strong>&quot;{query}&quot;</strong>.

<br /> Try checking for typos or using complete words.

</Typography>

</Paper>

</TableCell>

</TableRow>

);

}

TableNoData.propTypes = {

query: PropTypes.string,

}; import PropTypes from 'prop-types';

import Box from '@mui/material/Box';

import TableRow from '@mui/material/TableRow';

import Checkbox from '@mui/material/Checkbox';

import TableHead from '@mui/material/TableHead';

import TableCell from '@mui/material/TableCell';

import TableSortLabel from '@mui/material/TableSortLabel';

import { visuallyHidden } from './utils';

// ----------------------------------------------------------------------

export default function UserTableHead({

order,

orderBy,

rowCount,

headLabel,

numSelected,

onRequestSort,

onSelectAllClick,

}) {

const onSort = (property) => (event) => {

onRequestSort(event, property);

};

return (

<TableHead>

<TableRow>

<TableCell padding="checkbox">

<Checkbox

indeterminate={numSelected > 0 && numSelected < rowCount}

checked={rowCount > 0 && numSelected === rowCount}

onChange={onSelectAllClick}

/>

</TableCell>

{headLabel.map((headCell) => (

<TableCell

key={headCell.id}

align={headCell.align || 'left'}

sortDirection={orderBy === headCell.id ? order : false}

sx={{ width: headCell.width, minWidth: headCell.minWidth }}

>

<TableSortLabel

hideSortIcon

active={orderBy === headCell.id}

direction={orderBy === headCell.id ? order : 'asc'}

onClick={onSort(headCell.id)}

>

{headCell.label}

{orderBy === headCell.id ? (

<Box sx={{ ...visuallyHidden }}>

{order === 'desc' ? 'sorted descending' : 'sorted ascending'}

</Box>

) : null}

</TableSortLabel>

</TableCell>

))}

</TableRow>

</TableHead>

);

}

UserTableHead.propTypes = {

order: PropTypes.oneOf(['asc', 'desc']),

orderBy: PropTypes.string,

rowCount: PropTypes.number,

headLabel: PropTypes.array,

numSelected: PropTypes.number,

onRequestSort: PropTypes.func,

onSelectAllClick: PropTypes.func,

};

import PropTypes from 'prop-types';

import Tooltip from '@mui/material/Tooltip';

import Toolbar from '@mui/material/Toolbar';

import Typography from '@mui/material/Typography';

import IconButton from '@mui/material/IconButton';

import OutlinedInput from '@mui/material/OutlinedInput';

import InputAdornment from '@mui/material/InputAdornment';

import Iconify from 'src/components/iconify';

// ----------------------------------------------------------------------

export default function UserTableToolbar({ numSelected, filterName, onFilterName }) {

return (

<Toolbar

sx={{

height: 96,

display: 'flex',

justifyContent: 'space-between',

p: (theme) => theme.spacing(0, 1, 0, 3),

...(numSelected > 0 && {

color: 'primary.main',

bgcolor: 'primary.lighter',

}),

}}

>

{numSelected > 0 ? (

<Typography component="div" variant="subtitle1">

{numSelected} selected

</Typography>

) : (

<OutlinedInput

value={filterName}

onChange={onFilterName}

placeholder="Search user..."

startAdornment={

<InputAdornment position="start">

<Iconify

icon="eva:search-fill"

sx={{ color: 'text.disabled', width: 20, height: 20 }}

/>

</InputAdornment>

}

/>

)}

{numSelected > 0 ? (

<Tooltip title="Delete">

<IconButton>

<Iconify icon="eva:trash-2-fill" />

</IconButton>

</Tooltip>

) : (

<Tooltip title="Filter list">

<IconButton>

<Iconify icon="ic:round-filter-list" />

</IconButton>

</Tooltip>

)}

</Toolbar>

);

}

UserTableToolbar.propTypes = {

numSelected: PropTypes.number,

filterName: PropTypes.string,

onFilterName: PropTypes.func,

};

PROFILE:

/\* eslint-disable import/no-extraneous-dependencies \*/

/\* eslint-disable import/no-unresolved \*/

// eslint-disable-next-line

import { useRef, useState, useEffect } from 'react';

import Box from '@mui/material/Box';

import Card from '@mui/material/Card';

import Stack from '@mui/material/Stack';

import Divider from '@mui/material/Divider';

import TextField from '@mui/material/TextField';

import Typography from '@mui/material/Typography';

import LoadingButton from '@mui/lab/LoadingButton';

import { alpha, styled, useTheme } from '@mui/material/styles';

import { bgGradient } from 'src/theme/css';

import Logo from 'src/components/logo';

import MenuItem from '@mui/material/MenuItem';

// ----------------------------------------------------------------------

import Button from '@mui/material/Button';

import { getData, putFormData } from 'src/services/apiService';

import { usePathname, useRouter } from 'src/routes/hooks';

import { toast } from 'react-toastify';

import 'react-toastify/dist/ReactToastify.css';

// eslint-disable-next-line import/no-extraneous-dependencies

import DatePicker from 'react-datepicker';

import 'react-datepicker/dist/react-datepicker.css';

export default function ProfileView() {

const theme = useTheme();

const fileInputRef = useRef(null);

const pathname = usePathname();

const [selectedFile, setSelectedFile] = useState(null);

const [previewURL, setPreviewURL] = useState('');

const [userDetail, setUserDetail] = useState(null);

const [selectedDate, setSelectedDate] = useState(new Date());

const router = useRouter();

const StyledDatePicker = styled(DatePicker)({

width: '100%',

height: '9vh',

border: '1px solid #8080805e',

borderRadius: '0.8rem',

'& .react-datepicker-wrapper': {

width: '100%',

},

});

const [formData, setFormData] = useState({

name: '',

phoneNumber: '',

dob: new Date(),

gender: '',

pincode: '',

imageStatus: '',

latitude: '',

longitude: '',

});

const [formErrors, setFormErrors] = useState({

name: '',

phoneNumber: '',

dob: '',

gender: '',

pincode: '',

latitude: '',

longitude: '',

});

const handleDateChange = (date) => {

setFormData({ ...formData, "dob": date });

};

useEffect(() => {

const detail = localStorage.getItem('loginDetails');

if (!detail) {

Error('User details is null');

}

const userDetails = JSON.parse(detail);

setUserDetail(userDetails);

fetchUserData(userDetails.userId);

}, [pathname]);

const validateForm = () => {

let valid = true;

const errors = {};

if (!formData.name) {

errors.name = 'Name is required';

valid = false;

}

if (!formData.phoneNumber) {

errors.phoneNumber = 'Phone Number is required';

valid = false;

}

if (!formData.pincode) {

errors.pincode = 'Pincode is required';

valid = false;

}

if (!formData.latitude) {

errors.latitude = 'Latitude is required';

valid = false;

}

if (!formData.longitude) {

errors.longitude = 'Longitude is required';

valid = false;

}

if (!formData.gender) {

errors.gender = 'Gender is required';

valid = false;

}

setFormErrors(errors);

return valid;

};

const fetchUserData = async (userId) => {

const userDetail1 = await getData(`user/user/${userId}`);

if (userDetail1.image) {

const url = `data:image/png;base64,${userDetail1.image.data}`;

setPreviewURL(url);

}

setFormData({

name: userDetail1.name,

phoneNumber: userDetail1.phoneNumber,

pincode: userDetail1.pincode,

latitude: userDetail1.latitude,

longitude: userDetail1.longitude,

dob: userDetail1.dob ? userDetail1.dob : new Date(),

gender: userDetail1.gender,

imageStatus: 'Unchanged',

});

};

const handleChange = (event) => {

const { name, value } = event.target;

setFormData({ ...formData, [name]: value }); // Update the respective field in the formData state

};

const handleButtonClick = () => {

fileInputRef.current.click();

};

const handleFileInputChange = (e) => {

const file = e.target.files[0];

setSelectedFile(file);

setFormData({ ...formData, imageStatus: 'Added' });

const reader = new FileReader();

reader.onload = () => {

setPreviewURL(reader.result);

};

reader.readAsDataURL(file);

};

const handleUpdate = async () => {

const isValid = validateForm();

if (isValid) {

const formData1 = new FormData();

formData1.append('name', formData.name);

formData1.append('phoneNumber', formData.phoneNumber);

formData1.append('dob', formData.dob);

formData1.append('gender', formData.gender);

formData1.append('pincode', formData.pincode);

formData1.append('latitude', formData.latitude);

formData1.append('longitude', formData.longitude);

formData1.append('image', selectedFile);

formData1.append('userId', userDetail.userId)

formData1.append("imageStatus", formData.imageStatus);

try {

await putFormData('user/update', formData1);

router.push('/dashboard');

toast.success('Profile updated successfully');

} catch (error) {

console.error('Error registering user:', error);

toast.error('Failed to update profile');

}

} else {

toast.error('Please fill in all required fields correctly');

}

};

const genders = [

{

value: 'male',

label: 'Male',

},

{

value: 'female',

label: 'Female',

},

{

value: 'other',

label: 'Other',

},

];

const renderForm = (

<>

<Stack spacing={3}>

<TextField

name="name"

label="Name"

error={!!formErrors.name}

helperText={formErrors.name}

value={formData.name}

onChange={handleChange}

/>

<TextField

name="phoneNumber"

label="Phone Number"

value={formData.phoneNumber}

error={!!formErrors.phoneNumber}

helperText={formErrors.phoneNumber}

onChange={handleChange}

/>

<StyledDatePicker

selected={formData.dob}

onChange={handleDateChange}

dateFormat="dd/MM/yyyy"

maxDate={new Date()} // Restrict future dates

showYearDropdown

scrollableYearDropdown

yearDropdownItemNumber={15}

/>

<TextField

select

name="gender"

label="Gender"

value={formData.gender}

onChange={handleChange}

>

{genders.map((option) => (

<MenuItem key={option.value} value={option.value}>

{option.label}

</MenuItem>

))}

</TextField>

<TextField

name="pincode"

label="Pincode"

value={formData.pincode}

error={!!formErrors.pincode}

helperText={formErrors.pincode}

onChange={handleChange}

/>

<TextField

name="latitude"

label="Latitude"

value={formData.latitude}

error={!!formErrors.latitude}

helperText={formErrors.latitude}

onChange={handleChange}

/>

<TextField

name="longitude"

label="Longitude"

value={formData.longitude}

error={!!formErrors.longitude}

helperText={formErrors.longitude}

onChange={handleChange}

/>

</Stack>

<Stack direction="row" alignItems="center" justifyContent="flex-end" sx={{ my: 3 }} />

<LoadingButton

fullWidth

size="large"

type="submit"

variant="contained"

color="inherit"

onClick={handleUpdate}

>

Update

</LoadingButton>

</>

);

/\* name,number,email,password, re-password \*/

return (

<Box

sx={{

...bgGradient({

color: alpha(theme.palette.background.default, 0.9),

imgUrl: '/assets/background/overlay\_4.jpg',

}),

minHeight: '100%',

overflowY: 'auto',

}}

>

<Logo

sx={{

position: 'fixed',

top: { xs: 16, md: 24 },

left: { xs: 16, md: 24 },

}}

/>

<Stack alignItems="center" justifyContent="center" sx={{ height: 1 }}>

<Card

sx={{

p: 5,

width: 1,

maxWidth: 620,

}}

>

<Typography variant="h4">Profile Update</Typography>

<Divider sx={{ my: 3 }} />

<Stack direction="row" spacing={3}>

<input

type="file"

ref={fileInputRef}

style={{ display: 'none' }}

onChange={handleFileInputChange}

/>

<Button variant="outlined" onClick={handleButtonClick}>

Select Image

</Button>

</Stack>

<Divider sx={{ my: 3 }} />

<Stack direction="row" spacing={3}>

{previewURL && (

<div>

<img

src={previewURL}

alt="Preview"

style={{ maxWidth: '100%', maxHeight: '100px' }}

/>

</div>

)}

</Stack>

<Divider sx={{ my: 3 }} />

{renderForm}

</Card>

</Stack>

</Box>

);

}

**NODIFICATION:**

**/\* eslint-disable import/no-extraneous-dependencies \*/**

**/\* eslint-disable import/no-unresolved \*/**

**// eslint-disable-next-line**

**import { useState, useEffect } from 'react';**

**// ----------------------------------------------------------------------**

**import { getData, putData } from 'src/services/apiService';**

**import { usePathname } from 'src/routes/hooks';**

**import { toast } from 'react-toastify';**

**import 'react-toastify/dist/ReactToastify.css';**

**// eslint-disable-next-line import/no-extraneous-dependencies**

**import 'react-datepicker/dist/react-datepicker.css';**

**import { Table, TableBody, TableCell, TableRow, TableHead } from '@mui/material';**

**import Button from '@mui/material/Button';**

**export default function NotificationView() {**

**const pathname = usePathname();**

**const [data, setData] = useState([]);**

**const [userDetails, setUserDetails] = useState([]);**

**useEffect(() => {**

**const detail = localStorage.getItem('loginDetails');**

**if (!detail) {**

**Error('User details is null');**

**}**

**const userDetail = JSON.parse(detail);**

**setUserDetails(userDetail)**

**if (userDetail.role === "1") {**

**fetchFarmerData(userDetail.userId)**

**}**

**else {**

**fetchData(userDetail.userId);**

**}**

**}, [pathname]);**

**const handleButtonClick = async (status, notificationId) => {**

**// Call the function passed from the parent component**

**try {**

**await putData('notification/status/update', { notificationId, status });**

**toast.success('Status updated successfully');**

**if (userDetails.role === "1") {**

**fetchFarmerData(userDetails.userId)**

**}**

**else {**

**fetchData(userDetails.userId);**

**}**

**} catch (error) {**

**console.error('Error creating product:', error);**

**toast.error('Failed to create product');**

**}**

**};**

**const fetchData = async (userId) => {**

**const options = {**

**year: 'numeric',**

**month: 'long',**

**day: 'numeric',**

**hour: '2-digit',**

**minute: '2-digit',**

**second: '2-digit',**

**hour12: false,**

**timeZone: 'UTC'**

**};**

**try {**

**const response = await getData(`notification/list/${userId}`);**

**response.map(x => {**

**const dateTime = new Date(x.notificationDate);**

**const formattedDateTime = dateTime.toLocaleString('en-US', options);**

**x.notificationDate = formattedDateTime**

**return x;**

**})**

**const jsonData = response;**

**setData(jsonData);**

**} catch (error) {**

**console.error('Error fetching data:', error);**

**}**

**};**

**const fetchFarmerData = async (userId) => {**

**const options = {**

**year: 'numeric',**

**month: 'long',**

**day: 'numeric',**

**hour: '2-digit',**

**minute: '2-digit',**

**second: '2-digit',**

**hour12: false,**

**timeZone: 'UTC'**

**};**

**try {**

**const response = await getData(`notification/farmer/list/${userId}`);**

**response.map(x => {**

**const dateTime = new Date(x.notificationDate);**

**const formattedDateTime = dateTime.toLocaleString('en-US', options);**

**x.notificationDate = formattedDateTime**

**return x;**

**})**

**const jsonData = response;**

**setData(jsonData);**

**} catch (error) {**

**console.error('Error fetching data:', error);**

**}**

**};**

**/\* name,number,email,password, re-password \*/**

**return (**

**<Table>**

**<TableHead>**

**<TableRow>**

**<TableCell>S.No</TableCell>**

**<TableCell>Message</TableCell>**

**<TableCell>Date</TableCell>**

**<TableCell>Status</TableCell>**

**{userDetails.role === "1" ? (**

**<TableCell>Action</TableCell>**

**) : null}**

**</TableRow>**

**</TableHead>**

**<TableBody>**

**{data.map((row, index) => (**

**<TableRow key={row.id}>**

**<TableCell component="th" scope="row">**

**{index + 1} {/\* SNO starts from 1 \*/}**

**</TableCell>**

**<TableCell component="th" scope="row">**

**{row.message}**

**</TableCell>**

**<TableCell>{row.notificationDate}</TableCell>**

**<TableCell>{row.status}</TableCell>**

**{userDetails.role === "1" ? (**

**<TableCell>**

**<Button variant="contained" onClick={() => handleButtonClick("Accept", row.\_id)}>**

**Accept**

**</Button>**

**<Button style={{ backgroundColor: 'red', color: 'white', marginLeft: '10px' }} variant="contained" onClick={() => handleButtonClick("Decline", row.\_id)}>**

**Decline**

**</Button>**

**</TableCell>**

**) : null}**

**</TableRow>**

**))}**

**</TableBody>**

**</Table>**

**);**

**}**

**VIWE FARMER PRODUCT:**

**import { useEffect, useState } from 'react';**

**import Stack from '@mui/material/Stack';**

**import Container from '@mui/material/Container';**

**import Grid from '@mui/material/Unstable\_Grid2';**

**import Typography from '@mui/material/Typography';**

**import Button from '@mui/material/Button';**

**import { useNavigate } from 'react-router-dom';**

**import { getData } from 'src/services/apiService';**

**import ProductCard from '../product-card';**

**// ----------------------------------------------------------------------**

**export default function ProductsView() {**

**const [openFilter, setOpenFilter] = useState(false);**

**const [data, setData] = useState([]);**

**const [userDetail, setUserDetail] = useState(null);**

**const navigateTo = useNavigate();**

**const handleAddProductClick = () => {**

**navigateTo('/create-product');**

**};**

**useEffect(() => {**

**const detail = localStorage.getItem('loginDetails');**

**if (!detail) {**

**Error('User details is null');**

**}**

**const userDetails = JSON.parse(detail);**

**setUserDetail(userDetails);**

**fetchData(userDetails.userId);**

**}, []);**

**const fetchData = async (userId) => {**

**try {**

**let response = await getData(`product/list/${userId}`);**

**console.log('Response', response);**

**response = response.map((x) => {**

**const url = `data:image/png;base64,${x.image.data}`;**

**x.url = url;**

**return x;**

**});**

**const jsonData = response;**

**setData(jsonData);**

**} catch (error) {**

**console.error('Error fetching data:', error);**

**}**

**};**

**return (**

**<Container>**

**<Typography variant="h4" sx={{ mb: 5 }}>**

**Products**

**</Typography>**

**<Stack direction="row" spacing={2}>**

**<div style={{ textAlign: 'right', marginBottom: '20px' }}>**

**<Button variant="outlined" color="primary" onClick={handleAddProductClick}>**

**Add Product**

**</Button>**

**</div>**

**</Stack>**

**<Grid container spacing={3}>**

**{data.map((product) => (**

**<Grid key={product.\_id} xs={12} sm={6} md={3}>**

**<ProductCard product={product} />**

**</Grid>**

**))}**

**</Grid>**

**</Container>**

**);**

**}**

**VIWE USER PRODUCT:**

import PropTypes from 'prop-types';

import Box from '@mui/material/Box';

import Link from '@mui/material/Link';

import Card from '@mui/material/Card';

import Stack from '@mui/material/Stack';

import Typography from '@mui/material/Typography';

import { fCurrency } from 'src/utils/format-number';

import Label from 'src/components/label';

import { ColorPreview } from 'src/components/color-utils';

import Badge from '@mui/material/Badge';

// ----------------------------------------------------------------------

export default function ShopProductCard({ product }) {

const renderStatus = (

<Label

variant="filled"

color={(product.status === 'sale' && 'error') || 'info'}

sx={{

zIndex: 9,

top: 16,

right: 16,

position: 'absolute',

textTransform: 'uppercase',

}}

>

{product.status}

</Label>

);

const renderImg = (

<Box

component="img"

alt={product.name}

src={product.url}

sx={{

top: 0,

width: 1,

height: 1,

objectFit: 'cover',

position: 'absolute',

}}

/>

);

const renderPrice = (

<Typography variant="subtitle1">

<Typography

component="span"

variant="body1"

sx={{

color: 'text.disabled',

textDecoration: 'line-through',

}}

>

{product.priceSale && fCurrency(product.priceSale)}

</Typography>

&nbsp;

{fCurrency(product.price)}

</Typography>

);

return (

<Card>

<Box sx={{ pt: '100%', position: 'relative' }}>

{renderImg}

</Box>

<Stack spacing={2} sx={{ p: 3 }}>

<Link color="inherit" underline="hover" variant="subtitle2" noWrap>

{product.name}

</Link>

{product.description}

<Badge badgeContent={product.category || ''} color="primary"

style={{

left: '-10px', // Adjust the value as needed to move the badge to the desired position

top: '50%', // Center vertically

transform: 'translateY(-50%)', // Center vertically

}}

/>

<Stack direction="row" alignItems="center" justifyContent="space-between">

{renderPrice}

</Stack>

</Stack>

</Card>

);

}

ShopProductCard.propTypes = {

product: PropTypes.object,

};

ERROR:

import Box from '@mui/material/Box';

import Button from '@mui/material/Button';

import Container from '@mui/material/Container';

import Typography from '@mui/material/Typography';

import { RouterLink } from 'src/routes/components';

import Logo from 'src/components/logo';

// ----------------------------------------------------------------------

export default function NotFoundView() {

const renderHeader = (

<Box

component="header"

sx={{

top: 0,

left: 0,

width: 1,

lineHeight: 0,

position: 'fixed',

p: (theme) => ({ xs: theme.spacing(3, 3, 0), sm: theme.spacing(5, 5, 0) }),

}}

>

<Logo />

</Box>

);

return (

<>

{renderHeader}

<Container>

<Box

sx={{

py: 12,

maxWidth: 480,

mx: 'auto',

display: 'flex',

minHeight: '100vh',

textAlign: 'center',

alignItems: 'center',

flexDirection: 'column',

justifyContent: 'center',

}}

>

<Typography variant="h3" sx={{ mb: 3 }}>

Sorry, page not found!

</Typography>

<Typography sx={{ color: 'text.secondary' }}>

Sorry, we couldn’t find the page you’re looking for. Perhaps you’ve mistyped the URL? Be

sure to check your spelling.

</Typography>

<Box

component="img"

src="/assets/illustrations/illustration\_404.svg"

sx={{

mx: 'auto',

height: 260,

my: { xs: 5, sm: 10 },

}}

/>

<Button href="/" size="large" variant="contained" component={RouterLink}>

Go to Home

</Button>

</Box>

</Container>

</>

);

}

API BACK-END:

const express = require('express');

const cors = require('cors');

const loginRoutes = require('./routes/login');

const registerRoutes = require('./routes/register');

const userRoutes = require('./routes/user');

const productRoutes = require('./routes/product');

const notificationRoutes = require('./routes/notification');

const historyRoutes = require('./routes/order');

const { logger } = require('./middleware/loggerMiddleware');

const PORT = process.env.PORT || 3000;

const mongoose = require('mongoose');

const app = express();

app.use(express.json());

app.use(cors())

app.use(logger);

//app.use(authenticateToken);

const apiRouter = express.Router();

// Mount login and register routes

apiRouter.use('/login', loginRoutes);

apiRouter.use('/register', registerRoutes);

apiRouter.use('/user', userRoutes);

apiRouter.use('/product', productRoutes);

apiRouter.use('/order', historyRoutes);

apiRouter.use('/notification', notificationRoutes);

app.use('/api/v1', apiRouter);

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/farmer', {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => {

console.log('Connected to MongoDB');

})

.catch((error) => {

console.error('Error connecting to MongoDB', error);

});

app.listen(PORT, () => {

console.log(`Server is running on port ${PORT}`);

});

ROUTES:

REGISTER:

// registerRoutes.js

const express = require('express');

const router = express.Router();

const { User } = require('../models/user'); // Import the User model

// Handle user registration

router.post('/', async (req, res) => {

try {

// Create a new user based on the request body

console.log("request", req.body)

const user = new User(req.body);

console.log("request 1", user)

await user.save(); // Save the user to the database

res.status(201).send(user);

} catch (error) {

res.status(400).send(error);

}

});

module.exports = router;

LOG-IN:

const express = require('express');

const jwt = require('jsonwebtoken');

const { User } = require('../models/user');

const router = express.Router();

router.post('/', async (req, res) => {

const { email, password } = req.body;

const user = await User.findOne({ email });

if (!user) {

return res.status(404).json({ message: 'User not found' });

}

const token = jwt.sign({ userId: user.\_id, email: user.email }, 'secret\_key', { expiresIn: '1h' });

res.json({ token, userId: user.\_id, name: user.name, role: user.role });

});

module.exports = router;

USER:

const express = require('express');

const multer = require('multer');

const router = express.Router();

const { User } = require('../models/user'); // Import the User model

const storage = multer.memoryStorage();

const upload = multer({ storage });

router.put('/update', upload.single('image'), async (req, res) => {

const { name, phoneNumber, dob, gender, address, pincode, imageStatus, latitude, userId, longitude } = req.body;

console.log("req", req)

// Read image data from file

let image = {}

if (imageStatus == "Added") {

image = {

data: req.file.buffer,

contentType: req.file.mimetype

};

}

// Find the user by their ID

const user = await User.findById(userId);

if (!user) {

return res.status(404).send('User not found');

}

try {

// Save user profile data and image data

user.name = name;

user.phoneNumber = phoneNumber;

user.dob = dob;

user.gender = gender;

user.address = address;

user.pincode = pincode;

user.image = image;

user.latitude = latitude;

user.longitude = longitude;

await user.save();

res.send('User registered successfully');

} catch (error) {

console.error('Error registering user:', error);

res.status(500).send('Failed to register user');

}

});

router.get('/user/:userId', async (req, res) => {

try {

const userId = req.params.userId;

const user = await User.findById(userId);

const c = Object.assign({}, user);

c.image = null

console.log("user", c)

if (!user) {

return res.status(404).send('User not found');

}

console.log("user.image", user.image, Object.keys(user.image))

const userDataWithImage = {

\_id: user.\_id,

name: user.name,

email: user.email,

phoneNumber: user.phoneNumber,

dob: user.dob,

gender: user.gender,

address: user.address,

pincode: user.pincode,

latitude: user.latitude,

longitude: user.longitude,

image: (!user.image || Object.keys(user.image).length === 0) ? null : {

contentType: user.image.contentType,

data: user.image.data?.toString('base64') // Convert binary data to base64 string

}

};

res.json(userDataWithImage);

} catch (error) {

console.error('Error retrieving user:', error);

res.status(500).send('Failed to retrieve user');

}

});

module.exports = router;

NOTIFICATION:

// registerRoutes.js

const express = require('express');

const router = express.Router();

const { Notification } = require('../models/notification'); // Import the User model

const { Product } = require('../models/product');

// Handle user registration

router.get('/list/:userId', async (req, res) => {

try {

// Create a new user based on the request body

const userId = req.params.userId;

const notification = await Notification.find({ userId: userId })

console.log("notifd")

res.json(notification);

} catch (error) {

res.status(400).send(error);

}

});

router.get('/farmer/list/:farmerId', async (req, res) => {

try {

// Create a new user based on the request body

const farmerId = req.params.farmerId;

const products = await Product.find({ userId: farmerId })

const productIds = products.map(product => product.\_id);

console.log("productIds", productIds)

const notification = await Notification.find({ productId: { $in: productIds } })

res.json(notification);

} catch (error) {

res.status(400).send(error);

}

});

router.put('/status/update', async (req, res) => {

const { notificationId, status } = req.body;

const notification = await Notification.findById(notificationId);

if (!notification) {

return res.status(404).send('User not found');

}

notification.status = status

await notification.save();

res.send('Notification updated successfully');

})

module.exports = router;

PRODUCT:

// registerRoutes.js

const express = require('express');

const router = express.Router();

const { Product } = require('../models/product'); // Import the User model

const { User } = require('../models/user'); // Import the User model

const multer = require('multer');

const storage = multer.memoryStorage();

const upload = multer({ storage });

// Handle user registration

router.get('/list/:userId', async (req, res) => {

try {

const userId = req.params.userId;

// Create a new user based on the request body

console.log("request", req.body)

let productList = await Product.find({ userId: userId });

if (!productList) {

return res.status(404).send('Product not found');

}

let updatedProduct = map\_products(productList)

res.json(updatedProduct)

} catch (error) {

res.status(400).send(error);

}

});

router.get('/list', async (req, res) => {

try {

let productList = await Product.find();

if (!productList) {

return res.status(404).send('Product not found');

}

let updatedProduct = map\_products(productList)

res.json(updatedProduct)

} catch (error) {

res.status(400).send(error);

}

});

router.get('/search/:userId/:searchText', async (req, res) => {

try {

const searchText = req.params.searchText;

const userId = req.params.userId;

const productList = await findProductsNearUser(userId, searchText);

if (!productList) {

return res.status(404).send('Product not found');

}

let updatedProduct = map\_products(productList)

res.json(updatedProduct)

} catch (error) {

res.status(400).send(error);

}

});

const findProductsNearUser = async (userId, searchText) => {

const user = await User.findById(userId); // Assuming you have a User model

console.log("user", user)

const userLat = user.latitude;

const userLon = user.longitude;

const productList = await Product.find({ name: { $regex: searchText, $options: 'i' } });

const nearbyProducts = [];

for (const product of productList) {

const farmerId = product.userId;

const farmer = await User.findById(farmerId);

const distance = haversineDistance(userLat, userLon, farmer.latitude, farmer.longitude);

console.log("distance", distance)

if (distance <= 5) {

nearbyProducts.push(product);

}

}

return nearbyProducts;

};

const haversineDistance = (lat1, lon1, lat2, lon2) => {

const R = 6371; // Radius of the Earth in kilometers

const dLat = (lat2 - lat1) \* Math.PI / 180; // Convert degrees to radians

const dLon = (lon2 - lon1) \* Math.PI / 180;

const a = Math.sin(dLat / 2) \* Math.sin(dLat / 2) +

Math.cos(lat1 \* Math.PI / 180) \* Math.cos(lat2 \* Math.PI / 180) \*

Math.sin(dLon / 2) \* Math.sin(dLon / 2);

const c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

const distance = R \* c; // Distance in kilometers

return distance;

};

router.get('/:productId', async (req, res) => {

try {

const productId = req.params.productId;

let productList = await Product.find({ \_id: productId });

if (!productList) {

return res.status(404).send('Product not found');

}

let updatedProduct = map\_products(productList)

res.json(updatedProduct[0])

} catch (error) {

res.status(400).send(error);

}

});

function map\_products(productList) {

let updatedProduct = []

for (let product of productList) {

//console.log("product",product.image)

let productImage = (!product.image || Object.keys(product.image).length === 0) ? null : {

contentType: product.image.contentType,

data: product.image.data.toString('base64') // Convert binary data to base64 string

};

let p = {

\_id: product.id,

name: product.name,

description: product.description,

quantity1: product.quantity1,

quantity2: product.quantity2,

quantity3: product.quantity3,

category: product.category,

userId: product.userId,

image: productImage

}

updatedProduct.push(p)

}

return updatedProduct;

}

router.post('/create', upload.single('image'), async (req, res) => {

const { name, description, quantity1, quantity2, quantity3, imageStatus, category, userId } = req.body;

console.log("req", req)

// Read image data from file

let image = {}

if (imageStatus == "Added") {

image = {

data: req.file.buffer,

contentType: req.file.mimetype

};

}

// Find the user by their ID

const product = new Product()

if (!product) {

return res.status(404).send('User not found');

}

try {

// Save user profile data and image data

product.name = name;

product.description = description;

product.quantity1 = quantity1;

product.quantity2 = quantity2;

product.quantity3 = quantity3;

product.category = category;

product.image = image;

product.userId = userId

await product.save();

res.send('Product created successfully');

} catch (error) {

console.error('Error registering user:', error);

res.status(500).send('Failed to create product');

}

});

module.exports = router;

THE CORE APP.JS:

const express = require('express');

const cors = require('cors');

const loginRoutes = require('./routes/login');

const registerRoutes = require('./routes/register');

const userRoutes = require('./routes/user');

const productRoutes = require('./routes/product');

const notificationRoutes = require('./routes/notification');

const historyRoutes = require('./routes/order');

const { logger } = require('./middleware/loggerMiddleware');

const PORT = process.env.PORT || 3000;

const mongoose = require('mongoose');

const app = express();

app.use(express.json());

app.use(cors())

app.use(logger);

//app.use(authenticateToken);

const apiRouter = express.Router();

// Mount login and register routes

apiRouter.use('/login', loginRoutes);

apiRouter.use('/register', registerRoutes);

apiRouter.use('/user', userRoutes);

apiRouter.use('/product', productRoutes);

apiRouter.use('/order', historyRoutes);

apiRouter.use('/notification', notificationRoutes);

app.use('/api/v1', apiRouter);

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/farmer', {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => {

console.log('Connected to MongoDB');

})

.catch((error) => {

console.error('Error connecting to MongoDB', error);

});

app.listen(PORT, () => {

console.log(`Server is running on port ${PORT}`);

});

# **CHAPTER 8 SYSTEM TESTING AND VALIDATION**

The goal of testing is to unearth flaws. The goal of testing is to identify and eliminate any potential problems with a product. Assemblies, subassemblies, and final products can all be tested for functionality with this method. Software testing is the practise of putting a programme through its paces to make sure it won't crash or else behave badly under use. There are numerous kinds of examinations. There are various kinds of tests, and each one caters to a different need.

**UNIT TESTING:**

Unit testing involves testing individual components or units of code in isolation. The purpose is to verify that each unit functions correctly according to its design and specifications. Unit tests are typically written by developers and executed as part of the development process to ensure the reliability and correctness of individual code units.

**INTEGRATION TESTING:**

Integration testing focuses on testing the interactions and interfaces between different components or modules of a system. The goal is to verify that the integrated components work together as expected and that data is passed correctly between them. Integration tests help identify any issues related to interoperability, communication, or data exchange between system components.

**FUNCTIONAL TESTING:**

Functional testing evaluates the functionality of a system or application by testing its features and capabilities against specified requirements. Test cases are designed to validate that the system performs its intended functions correctly and meets user expectations. Functional testing verifies inputs, outputs, user interactions, and system behaviors to ensure that the software meets its functional requirements.

**SYSTEM TESTING:**

System testing involves testing the entire integrated system as a whole to verify that it meets specified requirements and performs as expected in its intended environment. System tests assess the system's overall functionality, performance, reliability, and usability across different scenarios and use cases. This type of testing is usually conducted after integration testing and before deployment to production.

**WHITE BOX TESTING:**

White box testing, also known as structural or glass box testing, involves testing the internal structure and logic of a software application. Test cases are designed based on knowledge of the system's code and architecture to validate its correctness, completeness, and efficiency. White box testing techniques include code coverage analysis, path testing, and branch coverage to ensure thorough testing of all code paths and conditions.

**BLACK BOX TESTING:**

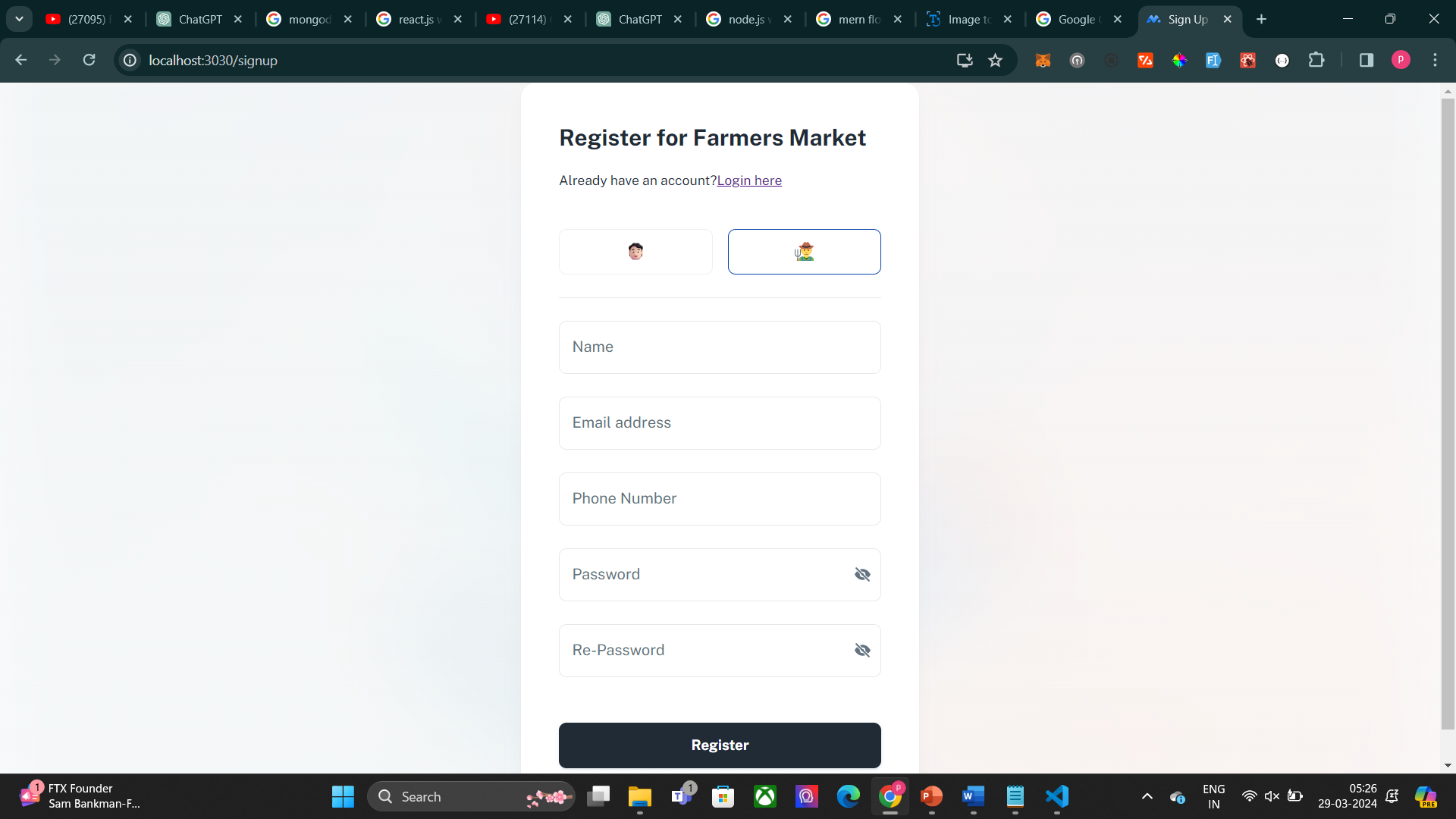
Black box testing focuses on testing the external behavior of a software application without knowledge of its internal implementation details. Test cases are designed based on functional specifications, requirements, and user inputs to validate the system's functionality from an end-user perspective. Black box testing techniques include equivalence partitioning, boundary value analysis, and use case testing to uncover defects and ensure that the system behaves as expected

**VALIDATION:**

Validation is the process of confirming whether a system, product, or service meets specified requirements and fulfills its intended purpose. It involves verifying that the final deliverable aligns with stakeholder expectations, complies with regulatory standards, and effectively addresses user needs. Validation encompasses reviewing functional and non-functional aspects, obtaining user feedback, ensuring regulatory compliance, and validating documentation to ensure accuracy and completeness. Ultimately, validation ensures that the end product meets quality standards, enhances user satisfaction, and mitigates risks throughout the development lifecycle

# **CHAPTER 9 SCREEN SHOTS**

**FARMER AND USER REGISTER PAGE:**

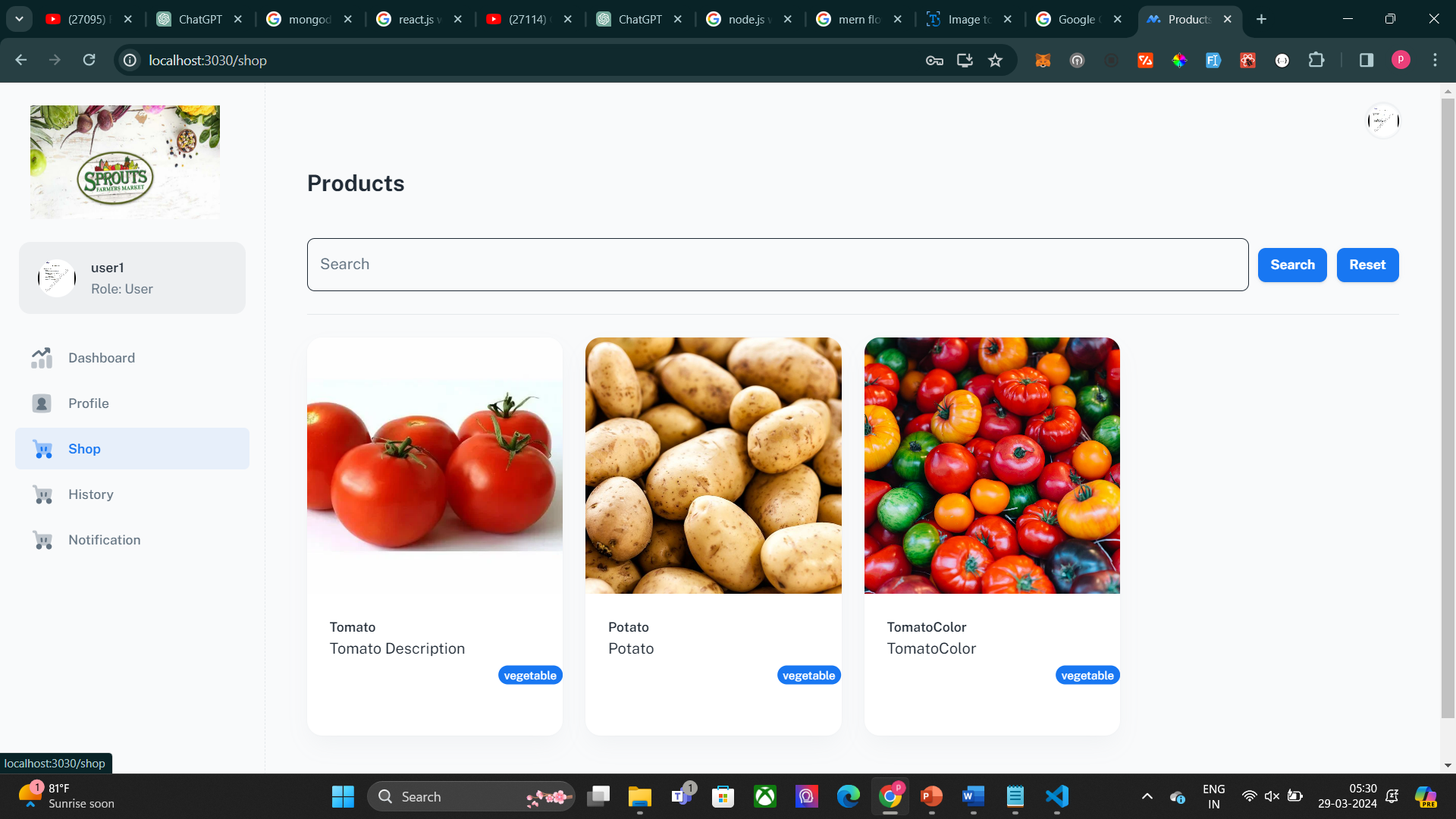


**LOG-IN PAGE:**

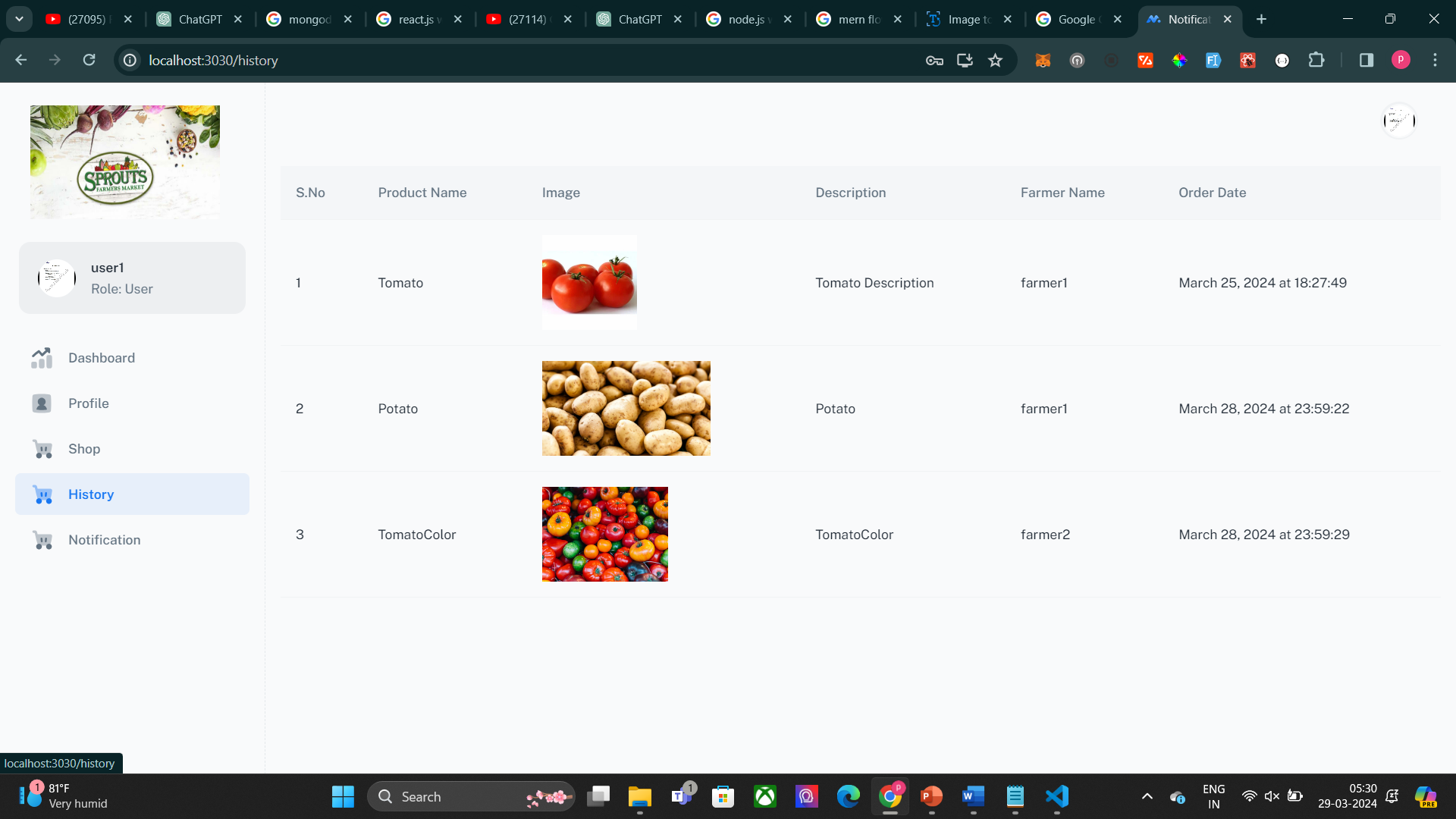
A screenshot of a computer

Description automatically generated

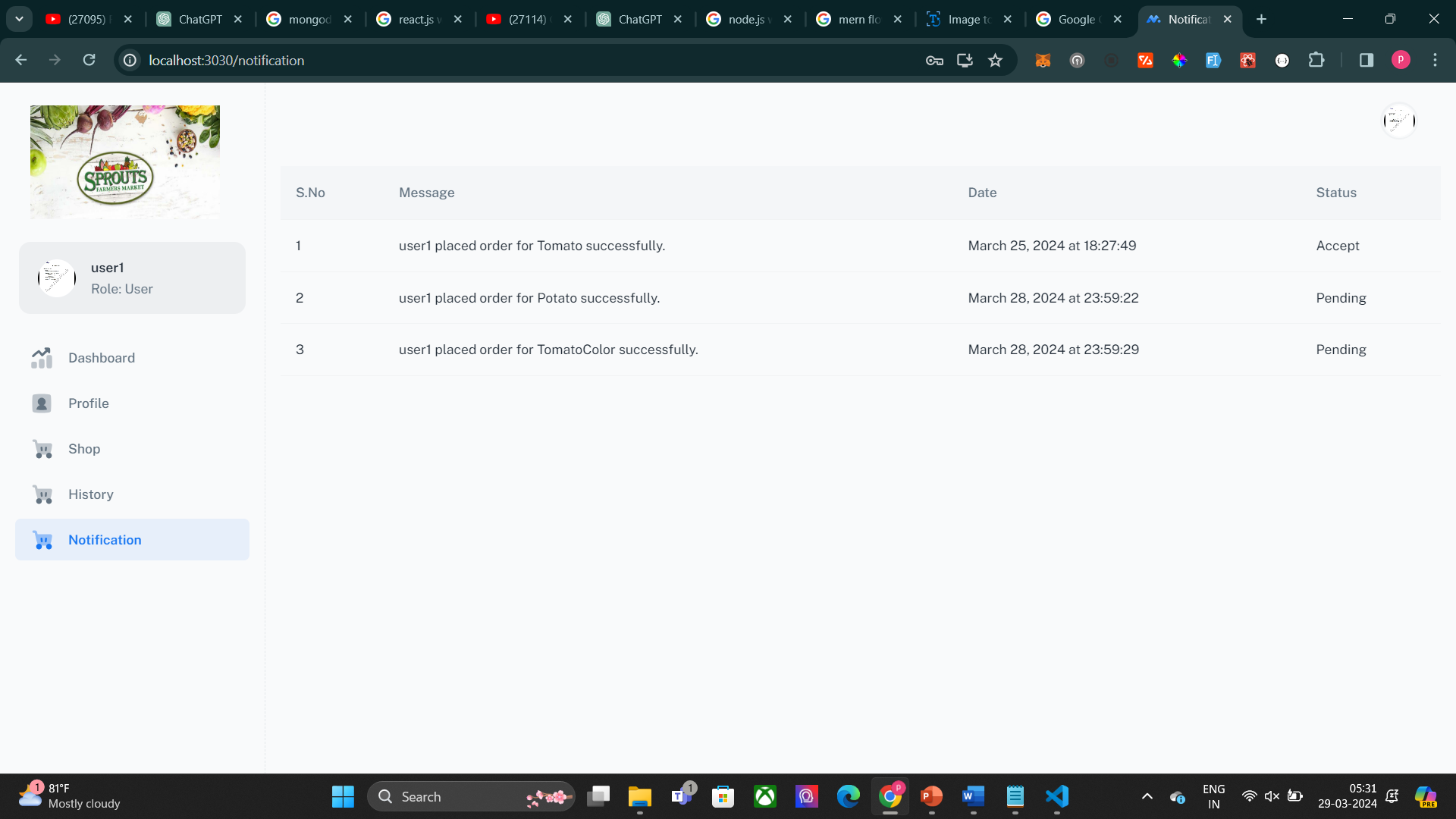
**USER PAGE:**



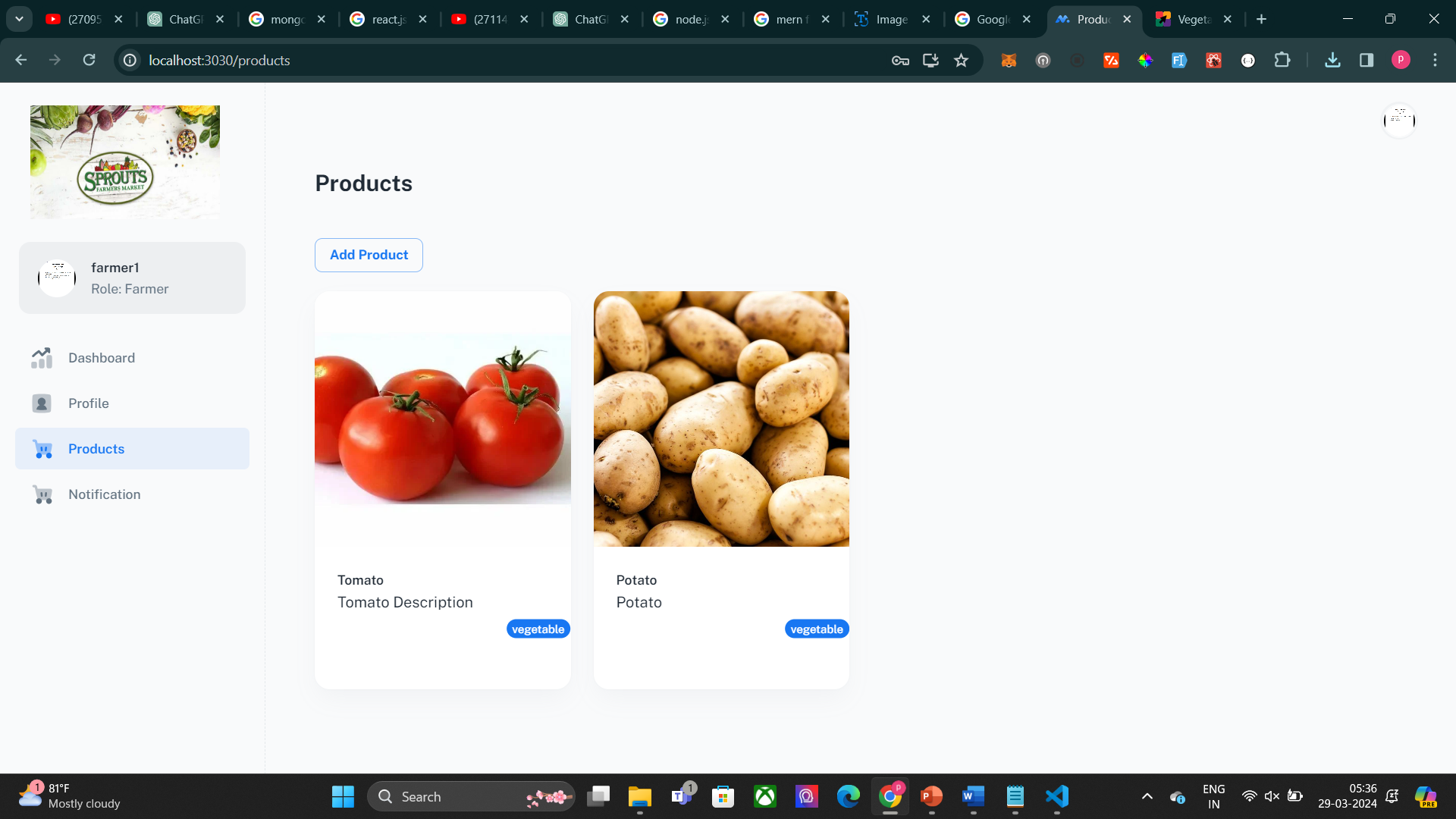
**USER HISTORY PAGE:**



**USER NODIFICATION PAGE:**



**FARMER PAGE:**



**FARMER NODIFICATION PAGE:**

A screenshot of a computer

Description automatically generated

# **CHAPTER 9 CONCLUTION AND FUTURE ENHANCEMENT**

Overall, the suggested Agriculture Project Marketplace and Local Product Search system signifies a substantial advancement in transforming the agriculture sector. This unique platform seeks to overcome the constraints of old methods and using contemporary technology to connect local farmers with consumers. Its goals include boosting community involvement, supporting sustainable agriculture, and encouraging the development of local farming projects.

The system incorporates the MERN stack to bring a variety of cutting-edge features, such as an interactive map, tailored product searches, efficient communication channels, real-time notifications, and responsive design. These features not only improve accessibility and user experience, but also enable smooth transactions and promote confidence between farmers and consumers. In addition, the Agriculture Project Marketplace and Local Product Search system guarantees the safety and dependability of user data and transactions by giving priority to security measures, conducting thorough testing, and deploying on a trustworthy cloud platform. This empowers both farmers and consumers in the digital era.

The Agriculture Project Marketplace and Local Product Search system exemplify the transformative potential of technology in promoting a connected, sustainable, and community-oriented agricultural environment. This platform fosters the development and involvement of individuals and groups with a vested interest, supports the growth of local economies, and advocates for the values of openness and long-term viability. As a result, it creates opportunities for a more positive and all-encompassing future in the field of agriculture.

**FUTURE ENHANCEMENT:**

**Mobile App Conversion:**

Converting the web application into a mobile app would increase accessibility and convenience for users, allowing them to access the platform from their smartphones or tablets. A mobile app could leverage device-specific features such as GPS for location-based services, push notifications for real-time updates, and offline capabilities for users in areas with limited connectivity. The app could provide a seamless and intuitive user experience tailored to mobile devices, enhancing engagement and adoption among farmers and consumers.

# **CHAPTER 10 BIBLIOGRAPHY**

**REFERENCES:**

* [1] Acemoglu, D., Robinson, J.A., Johnson, S., 2002. Reversal of Fortune : Geography and Institutions in the Making of the Modern World Income Distribution. Quarterly Journal of Economics 117, 1231–1294.
* [2] Africa Transport Policy Program, 2022. Third Development Plan - Completion report. The World Bank, Washington, D.C.
* [3] Aggarwal, S., 2018. Do rural roads create pathways out of poverty? Evidence from India. Journal of Development Economics 133, 375–395.
* [4] Aggarwal, S., Giera, B., Jeong, D., Robinson, J., Spearot, A., 2022. Market Access, Trade Costs, and Technology Adoption: Evidence from Northern Tanzania. Review of Economics and Statistics .
* [5] Allen, T., Atkin, D., Cantillo, S., Hernandez, C., 2020. Trucks. Working Paper
* [6] Asher, S., Novosad, P., 2020. Rural roads and local economic development.
* [7] Bergquist, L.F., McIntosh, C., Startz, M., 2022. Search Costs, Intermediation, and Trade: Experimental
* [8] Goland, C., and Bauer, S. (2004). When the apple falls close to the tree: local food systems and the preservation of diversity. Renew. Agric. Food Syst. 19, 228–236.
* [9] Gómez Sosa, L., and Arellanes Cancino, N. (2018). Del huerto al mercado: especies vegetales y comerciantes en la Villa de Zaachila, Oaxaca, Mexico. Rev. Mex. Agroecosistemas. 5, 43–54.

**WEB REFERENCES:**

JAVASCRIPT: <https://developer.mozilla.org/en-US/>

MONGODB: <https://www.mongodb.com/>

EXPRESS.JS FRAMWORK: <https://expressjs.com/>

RECT.JS: https://legacy.reactjs.org/

NODE.JS: <https://nodejs.org/en>

W3SCHOOL: https://www.w3schools.com/