ONLINE FOOD ORDERING PLATFORM

A PROJECT REPORT

Submitted by

AMIYADEVI S

(Reg. No: 24MCR002)

ELAKKIYA D

(Reg. No: 24MCR022)

JOTHISANKAR E

(Reg. No: 24MCR047)

in partial fulfilment of the requirements

for the award of the degree

of

MASTER OF COMPUTER APPLICATIONS

DEPARTMENT OF COMPUTER APPLICATIONS

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE – 638 060

DECEMBER 2024

DEPARTMENT OF COMPUTER APPLICATIONS

KONGU ENGINEERING COLLEGE

(Autonomous)

PERUNDURAI, ERODE – 638 060

December 2024

BONAFIDE CERTIFICATE

This is to certify that the project report entitled **"ONLINE FOOD ORDERING PLATFORM "** is the bonafide record of project work done by **AMIYADEVI S (24MCR002), ELAKKIYA D (24MCR022)** and **JOTHISANKAR E (24MCR047)** in partial fulfilment of the requirements for the award of the Degree of Master of Computer Applications of Anna University, Chennai during the year 2024-2025.

SUPERVISOR **HEAD OF THE DEPARTMENT**

(Signature with seal)

Date:

Submitted for the end semester viva voce examination held on _____

INTERNAL EXAMINER **EXTERNAL EXAMINER**

DECLARATION

I affirm that the project report titled **"ONLINE FOOD ORDERING PLATFROM"** being submitted in partial fulfilment of the requirements for the award of Master of Computer Applications is the original work carried out by us. It has not formed the part of any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

DATE:

AMIYADEVI S

(Reg. No: 24MCR002)

ELAKKIYA D

(Reg. No: 24MCR022)

JOTHISANKAR E

(Reg. No. 24MCR047)

I certify that the declaration made by the above candidates is true to the best of my knowledge.

Date: **Name and Signature of the Supervisor with seal**

ABSTRACT

One of the most significant innovations is the online food ordering platform, which allows customers to browse menus, place orders, and make payments from the convenience of their homes. This abstract focuses on the design and development of an Online Food Ordering Platform using the MERN stack—MongoDB, Express.js, React.js, and Node.js. The MERN stack is a popular combination of technologies that allows for the creation of dynamic, scalable, and high-performance web applications.This system is the use of MongoDB, a NoSQL database, to store and manage data such as customer profiles, restaurant menus, and order histories. MongoDB's flexibility in storing structured and unstructured data makes it an ideal choice for managing varied food items, customer details, and transaction records in real-time. Express.js, a minimal and flexible Node.js web application framework, is employed for creating the back-end APIs. These APIs handle essential operations like user authentication, order management, and payment processing, ensuring smooth interaction between the front-end and the database.

The front-end of the platform is built using React.js, a powerful JavaScript library for building user interfaces. React allows for the development of a highly interactive and dynamic user experience by updating the user interface (UI) in response to changes in data. Customers can easily browse restaurant menus, customize orders, and track their deliveries in real-time. React's component-based architecture enables modular design, allowing the platform to scale efficiently as new features are added, such as reviews, ratings, or loyalty programs.Node.js is used as the server-side runtime environment, enabling asynchronous and event-driven processing. This feature is particularly beneficial for handling high-volume, real-time operations such as order placements, payment transactions, and live updates on order statuses. Node.js also ensures the scalability of the platform, allowing it to handle large numbers of users and simultaneous requests, making it suitable for businesses with growing customer bases.

ACKNOWLEDGEMENT

We express our sincere thanks to our beloved correspondent **Thiru.A.K.ILNGO B.Com.,M.B.A.,LLB.,** and other philanthropic trust members of Kongu Vellalar Institute of Technology Trust for having provided with necessary resources to complete this project. We are always grateful to our beloved visionary Principal **Dr.v.BALUSAMYB.E.(Hons)., M.Tech., ph.D.,** and thank him for his motivation and moral support.

We express our deep sense of gratitude and profound thanks to **Dr.A.TAMILRASI M.Sc.,M.Phil.,Ph.D.,** Professor and Head of the Department in Computer Application for her invaluable commitment and guidance for this project.

We also like to express our gratitude and sincere thanks to our project coordinators **Dr.T.KAVITHA MCA.,** Assisant Professor(**SrG**), Department of Computer Applications, Kongu Engineering College who have motivated us in all aspects for completing the project in scheduled time.

We would like to express our gratitude and sincere thanks to our project guide **Mrs.T.Kalpana MCA.,PhD.,** Assistant Professor(**SrG**),Department of Computer Applications, Kongu Engineering College for giving her valuable guidance and suggestions which helped us in the successful completion of the project.

We owe a great deal of gratitude to our parents for helping us to overwhelmed in all proceedings. We bow our heart with heartfelt thanks to all those who thought us their warm services to succeed and achieve our work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 ABOUT THE PROJECT

1.1.1 Overview

The Online Food Ordering Platform is a web-based application that allows users to browse menus from various restaurants, place food orders, and track the status of their orders in real-time. It provides a seamless experience for both customers and restaurant owners, facilitating easy access to restaurant menus and management of food orders. This platform is built using the **MERN stack**, which includes **MongoDB**, **Express.js**, **React.js**, and **Node.js**.

1.1.2 Background

In today's fast-paced world, the demand for online food ordering platforms has significantly increased. Consumers increasingly prefer the convenience of browsing restaurant menus, placing orders, and having food delivered to their doorstep from the comfort of their homes or offices. The need for an efficient system that allows restaurants to handle orders, maintain up-to-date menus, and streamline communication with customers is evident. In addition, for customers, a seamless and user-friendly interface is critical to making the food ordering process quick and easy.

1.1.3 Problem Description

With the rise in demand for food delivery services, traditional food ordering methods are becoming less efficient and prone to errors. In the past, customers had to call restaurants directly, often facing long wait times or miscommunications. Restaurant staff had to manually handle incoming orders, update menus, and track deliveries, which often led to inefficiencies and customer dissatisfaction.

As the food delivery industry continues to grow, there is an increasing need for **digital solutions** that can streamline the ordering process, enhance customer experience, and improve restaurant operations. However, many small and medium-sized restaurants still struggle with building or adopting an online ordering system due to the costs, complexity, and lack of technical expertise.

1.2 EXISTING SYSTEM

The existing system is inconvenient for customer needing to have a physical copy of the menu, its time consuming, there is lack of visual confirmation that the order was placed correctly or not, Restaurants have to have an employee answering the phone and taking orders all the time which increases manual work and paper work. And there is also a huge difficulty in tracking customers past history and lack of data security.

The current System of a company is very ancient and need to be replacing as companies business is expanding. One of the biggest disadvantages of the current system is that lacking of computerized food order each time a customer need to order food he has to wait for the waiter to take their order and then give to the chef who will be preparing our food. So this process is very time consuming and very ancient.

1.3 DRAWBACKS OF EXISTING SYSTEM

While existing online food ordering systems have revolutionized the food delivery industry, they come with several limitations and challenges. Below are the key drawbacks of these existing systems:

- High Commission Fees
- Lack of Control Over Customer Data
- Branding and Customer Experience Limitations
- Delivery Dependency and Quality Control
- Increased Competition
- High Development and Maintenance Costs
- Limited Control Over Customer Interaction

1.4 PROPOSED SYSTEM

In the proposed system Security of data is provided where data are well protected for personal use and also ensures data accuracy during order placement process. It minimizes manual data entry. Since the data processing is very fast it provides great efficiency. This proposed system is user friendly and provides interactive interface with provision for customer to view menus.

The benefit of this is that if there is rush in the Restaurant then there will be chances that the waiters will be unavailable and the users can directly order the food to the chef online by using this application. The user will be given a username and a password, by sing that every time a user logs in. This implies that the customer is the regular user of the Restaurant.

1.5 ADVANTAGES OF PROPOSED SYSTEM

- As it is online the customer doesn't have to wait for the waiter to take their order and doesn't us to wait for the food as well.
- Waiters don't have to manually keep a record of all the food ordered by the customer and that work is very easy.
- Waiters don't have to manually calculate the amount of money to be paid by the customer after having food it is automatically done in the software.
- Records are maintained in computers so there are less chances of damage and loss of data

In the context of your **Online Food Ordering Platform** built using the **MERN stack**, the **anticipated conclusions** are based on the objectives of developing a more effective, scalable, and user-friendly system for restaurants and customers.

The proposed system is expected to significantly enhance the operational efficiency of restaurants by automating and streamlining processes such as **order management**, **inventory tracking**, and **customer interaction**. With features like real-time order tracking, dynamic menu updates, and integrated analytics, restaurants will be able to **reduce manual work**, minimize errors, and make data-driven decisions. The **flexible delivery management system** will also help ensure that restaurants can optimize delivery routes and manage their own or third-party delivery networks efficiently.

SUMMARY

This Chapter sets the foundation for the Online Food Ordering Platform project by defining its goals, structure, and the issues it aims to address. It introduces the core technologies (MERN stack) and outlines the expected impact of the platform on the food service industry. The chapter highlights the need for a scalable, user-friendly solution that offers both operational benefits for restaurants and enhanced experiences for customers. The platform is positioned as a cost-effective, efficient, and scalable solution to modernize the online food ordering process.

CHAPTER 2

SYSTEM ANALYSIS

2.1 PROBLEM IDENTIFICATION

An Online Food Ordering Platform is essential in today's fast-paced world, where convenience, efficiency, and a seamless customer experience are paramount. However, existing platforms face numerous challenges that hinder their effectiveness. The identification of these problems is a crucial step in designing a better system to address the pain points faced by both customers and restaurants. Restaurants often do not have direct access to customer data or the ability to communicate with their customers effectively. Many online food ordering platforms provide a suboptimal user experience, which can lead to customer frustration, cart abandonment, and lower conversion rates. Managing orders efficiently, including the preparation time and delivery process, is crucial for any food ordering platform. However, many existing systems fail to deliver timely and accurate orders.

2.2 FEASIBILITY STUDY

After doing the project Online Wedding Planner, study and analyzing all the existing or required functionalities of the system, the need task is to do the feasibility study for the project. All projects are feasible given unlimited resources and infinite time. Feasibility study includes consideration of all the possible ways to provide a solution to the given problem. The proposed solution should satisfy all the user requirements and should be flexible enough so that future changes can be easily done based on the future upcoming requirements.

A feasibility study is crucial for determining whether an online food ordering platform can be successfully developed and operated. The study evaluates the technical, economic, operational, and legal aspects of the project, ensuring that it is practical and sustainable in the long term.

2.2.1 Economic Feasibility

This is a very important aspect to be considered while developing a project. We decided the technology based on minimum possible cost factor. **Economic feasibility** analyzes whether the platform can be developed and maintained within budget and whether it can generate sufficient revenue.

- All hardware and software cost has to be borne by the organization.
- Overall we have estimated that the benefits the organization is going to receive from the proposed system will surely overcome the initial costs and the later on running cost for system.

2.2.2 Operational Feasibility

Operational feasibility evaluates whether the platform can be operated successfully and sustainably in the long term. A smooth, responsive, and intuitive UI/UX design will enhance the user experience for customers, ensuring ease of use while placing orders, browsing menus, and making payments. The platform will offer an intuitive dashboard for restaurant owners, allowing them to manage their menu, view orders, track deliveries, and access customer insights. Providing customer support via multiple channels, such as in-app chat, email, or a helpline, will help address user issues and complaints promptly. The platform can also efficiently handle delivery integration, either by leveraging existing services or developing a proprietary system.

2.2.3 Technical feasibility

Technical feasibility assesses whether the technology and resources needed to develop the platform are available, and whether the project can be implemented within the given constraints. This included the study of function, performance and constraints that may affect the ability to achieve an acceptable system. For this feasibility study, we studied complete functionality to be provided in the system as described in the System Requirement Specification (SRS), and checked if everything was possible using different type of frontend and backend platforms. The platform can scale effectively to handle increasing users and restaurant listings.

2.3 SYSTEM REQUIREMENT ANALYSIS

System Requirement Analysis (SRA) is the first and most crucial step in the software development lifecycle. It defines the needs and expectations of the stakeholders and ensures that all system functionalities align with the business objectives. In the case of an Online Food Ordering Platform, this analysis covers both functional and non-functional requirements that define how the platform will operate, interact with users, and meet performance standards.

2.3.1 Functional Requirements

Functional requirements define the core behaviors and features that the system must support. These are the critical functions that the platform must perform to meet the needs of its users (customers, restaurant owners, and delivery personnel).

2.3.1.1 User Registration and Authentication

User Sign-Up/Sign-In: The platform should allow users (customers, restaurant owners, delivery agents) to create accounts with basic credentials like email and password.

Customer: Can browse restaurants, place orders, and track deliveries.

Restaurant Owner: Can manage the menu, view orders, and update their restaurant's information.

Admin: Can oversee the entire system, approve new restaurants, manage users, and generate reports.

2.3.1.2 Restaurant Management

Restaurant Profile: Restaurant owners should be able to create and manage their restaurant profile, including uploading a logo, setting business hours, and providing contact details.

Menu Management: Restaurant owners should be able to create, edit, and delete menu items (with prices, descriptions, and images).

Order Management: Restaurant owners should receive real-time notifications of new orders, be able to accept or reject orders, and track the status of each order (e.g., preparing, out for delivery, completed).

2.3.1.3 Customer Interface and Experience

Browse and Search: Customers should be able to search for restaurants by type of cuisine, location, ratings, or name. The platform should allow browsing of restaurant menus with filter options.

Order Placement: Customers can select items, customize orders (e.g., special instructions for a dish), and add them to their cart. They should be able to review the cart and proceed to checkout.

Payment Integration: The platform must support multiple payment options such as credit/debit cards.

2.3.2 Non-Functional Requirements

Non-functional requirements describe how the system performs under certain conditions, ensuring that the platform provides a good user experience, security, scalability, and performance.

2.3.2.1 Performance Requirements

- **Response Time**: The platform should ensure a quick response time for user interactions. For instance, order placement should not take more than 2-3 seconds, and restaurant menus should load within 1 second.
- **System Availability**: The platform should be 99.9% available, meaning it should have minimal downtime. The system must handle high traffic during peak times, like lunch or dinner hours.

2.3.2.2 Scalability Requirements

- **Horizontal Scaling**:

  The system should be designed to scale horizontally. For example, additional servers should be able to be added to handle increased load as the number of users grows.

- **Database Scalability**: The MongoDB database must be scalable, especially as the number of users, restaurants, and orders increase. Features like sharding can be used to distribute data across multiple servers.

2.4 SOFTWARE DESCRIPTION

The Online Food Ordering Platform is a web-based application designed to connect customers with restaurants for the purpose of ordering food online. Built using the MERN stack (MongoDB, Express.js, React.js, Node.js), this platform will allow users to browse menus, place orders, make payments, and track their food delivery in real-time. It will also offer restaurant owners the ability to manage their menu, process orders, and view customer feedback, while providing delivery personnel with the tools to accept deliveries

2.4.1 MongoDB (Database)

MongoDB is a NoSQL database that stores data in a JSON-like format called BSON. It is highly scalable and flexible, making it an ideal choice for this platform, where we need to manage large amounts of data with a dynamic schema.

- **Primary Uses**:
  - Storing user profiles (customer, restaurant owner, admin, delivery agent).
  - Managing menu items for each restaurant.
  - Storing orders and their associated data (order details, payment status, delivery status).
  - Saving reviews and ratings for both restaurants and delivery services.

2.4.2 Express.js (Backend Framework)

Express.js is a lightweight, unopinionated web framework for Node.js that provides a set of tools and middleware for building web applications and APIs. It serves as the server-side platform that handles routing and manages API endpoints.

- **Primary Uses**:
    - Handling HTTP requests (GET, POST, PUT, DELETE) for CRUD operations (Create, Read, Update, Delete).
    - Managing user authentication and authorization with JWT (JSON Web Tokens).
    - Handling order placement, order tracking, and payment processing via APIs.
    - Connecting with MongoDB to store/retrieve user, order, and restaurant data.

2.4.3 React.js (Frontend Framework)

React.js is a JavaScript library for building dynamic, single-page applications (SPA). It is used to develop the user interface (UI), ensuring a fast and smooth experience for customers, restaurant owners, and delivery agents.

- **Primary Uses**:
    - **Dynamic UI Components**: Building reusable components for various platform sections (restaurant listing, menu display, order cart, and tracking).
    - **State Management**: Handling user interactions and managing the application's state using React hooks or Redux (for larger apps).
    - **Form Handling**: Managing form inputs like user sign-up/sign-in, order placement, and payment.

2.4.4 Node.js (Backend Environment)

Node.js is a JavaScript runtime environment that allows developers to execute JavaScript code outside the browser. It is fast and lightweight, making it an ideal choice for building scalable, event-driven applications.

- **Primary Uses**:
    - **Server-Side Logic**: Handling requests to interact with the database, process business logic (e.g., order management), and send responses to the front end.
    - **Handling Real-Time Communication**: Works with Socket.io to manage real-time order status and delivery tracking.

SUMMARY

This Chapter, System Requirement Analysis, focuses on detailing the requirements necessary to develop the Online Food Ordering Platform using the MERN stack (MongoDB, Express.js, React.js, and Node.js). It covers both functional and non-functional requirements of the system, which will guide the development process and ensure that the platform meets the needs of its users, including customers, restaurant owners, delivery agents, and administrators.

CHAPTER 3

SYSTEM DESIGN

3.1 SYSTEM MODIFICATION

Systems design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Systems design could be seen as the application of systems theory to product development. There is some overlap with the disciplines of systems analysis, systems architecture and systems engineering. The architectural design of a system emphasizes on the design of the systems architecture which describes the structure, behavior, and more views of that system.

The design activity often results in three separate outputs -

- Architecture design.
- High level design.
- Detailed design

3.2 ARCHITECTURAL DESIGN

Architectural design, is a set of principles- a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by solutions to recurring problems. You can think of architecture styles and patterns as sets of principles that shape an application.

3.3 THREE TIER ARCHITECTURE

Three architecture having three layers. They are

1. Client layer

2. Business layer

3. Data layer

3.3.1Client layer:

Here we design the form using textbox, label etc.

3.3.2 Business layer:

It is the intermediate layer which has the functions for client layer and it is used to make communication faster between client and data layer. It provides the business processes logic and the data access.

Data layer:

it has the database.

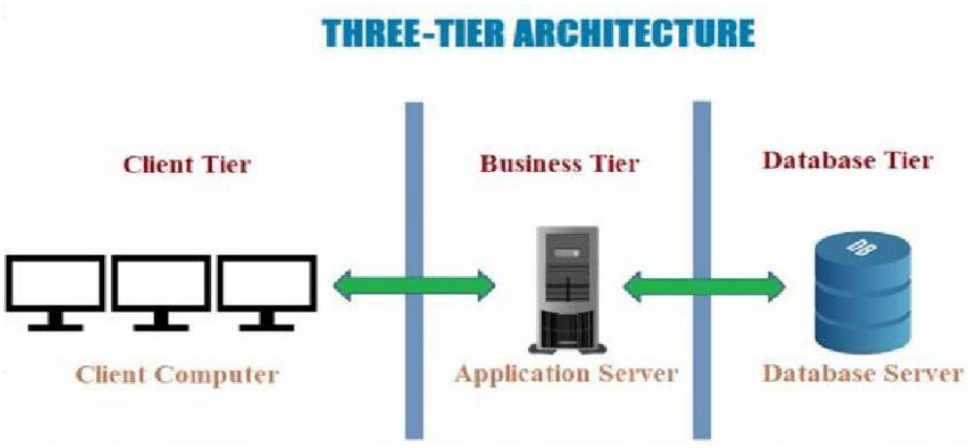

Figure 3.0 Three -Tier Architecture Diagram

Advantage

1. Easy to modify without affecting other modules

2. Fast communication

3. Performance will be good in three tier architecture.

3.4 HIGH LEVEL DESIGN

A high-level design provides an overview of a solution, platform, system, product, service, or process. Such an overview is important in a multi-project development to make sure that each supporting component design will be compatible with its neighboring designs and with the big picture. A high-level design document will usually include a high-level architecture diagram depicting the components, interfaces and networks that need to be further specified or developed. The document may also depict or otherwise refer to work flows and/or data flows between component systems.

3.5 DATA FLOW DIAGRAMS

A DFD shows what kind of information will be input to and output from the system, where the data will come from and go to, and where the data will be stored. It does not show information about the timing of process or information about whether processes will operate in sequence or in parallel. A Data Flow Diagram (DFD) is a graphical representation of the "flow" of data through an information system, modeling its process aspects. A DFD is often used as a preliminary step to create an overview of the system, which can later be elaborated. DFDs can also be used for the visualization of data processing.

External Entity

An external entity can represent a human, system or subsystem. It is where certain data comes from or goes to. It is external to the system we study, in terms of the business process. For this reason, people used to draw external entities on the edge of a diagram.

Process

A process is a business activity or function where the manipulation and transformation of data takes place. A process can be decomposed to finer level of details, for representing how data is being processed within the process

Data Store

A data store represents the storage of persistent data required and/or produced by theprocess. Here are some examples of data stores: membership forms, database table, etc.

Data Flow

A data flow represents the flow of information, with its direction represented by an arrow head that shows at the end(s) of flow connector.

3.5.1 ZERO-LEVEL DFD

It is also known as context diagram. Its designed to be an abstraction view showing the system as a single process with its relationship to its external entities. It represents the entire system as a single bubble with input and output data by incoming and outgoing arrows.

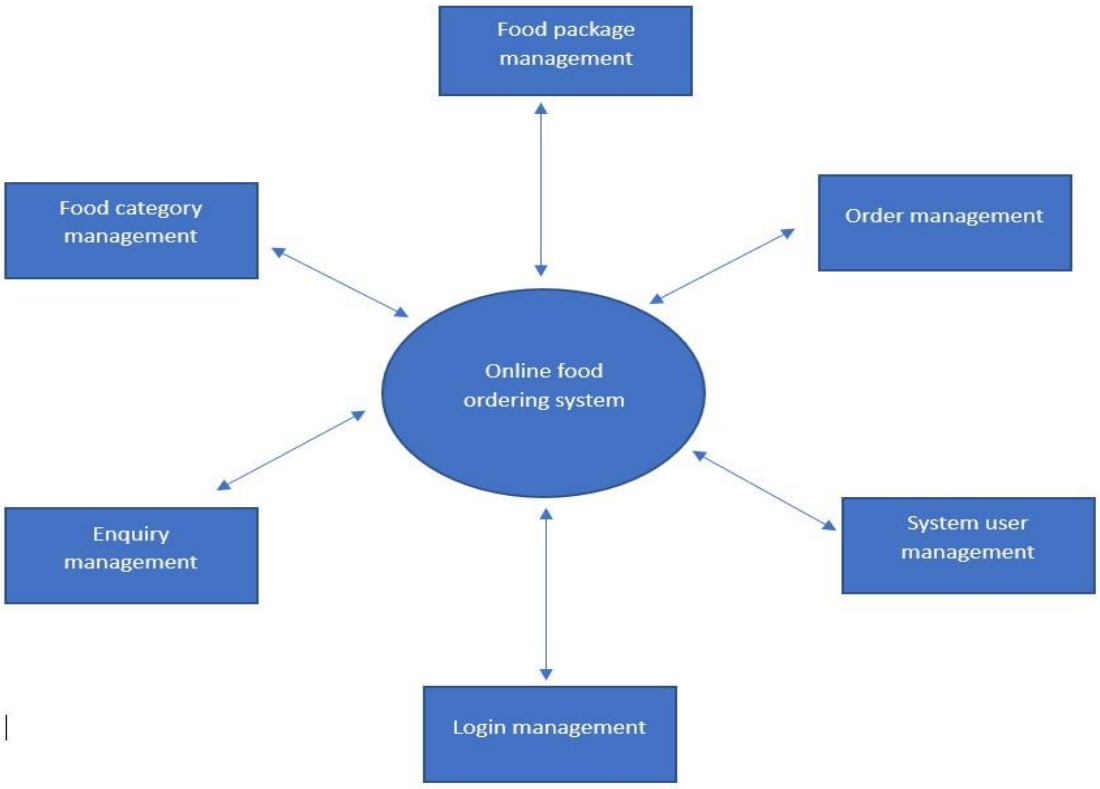

Figure 3.1 0th Level DFD diagram-online food ordering platform

3.5.2 FIRST-LEVEL DFD

This level shows all processes at the first level of numbering, data, stores, external entities and data flows between them. The purpose of this level is to show the major and high level processes of the system and their interrelation.

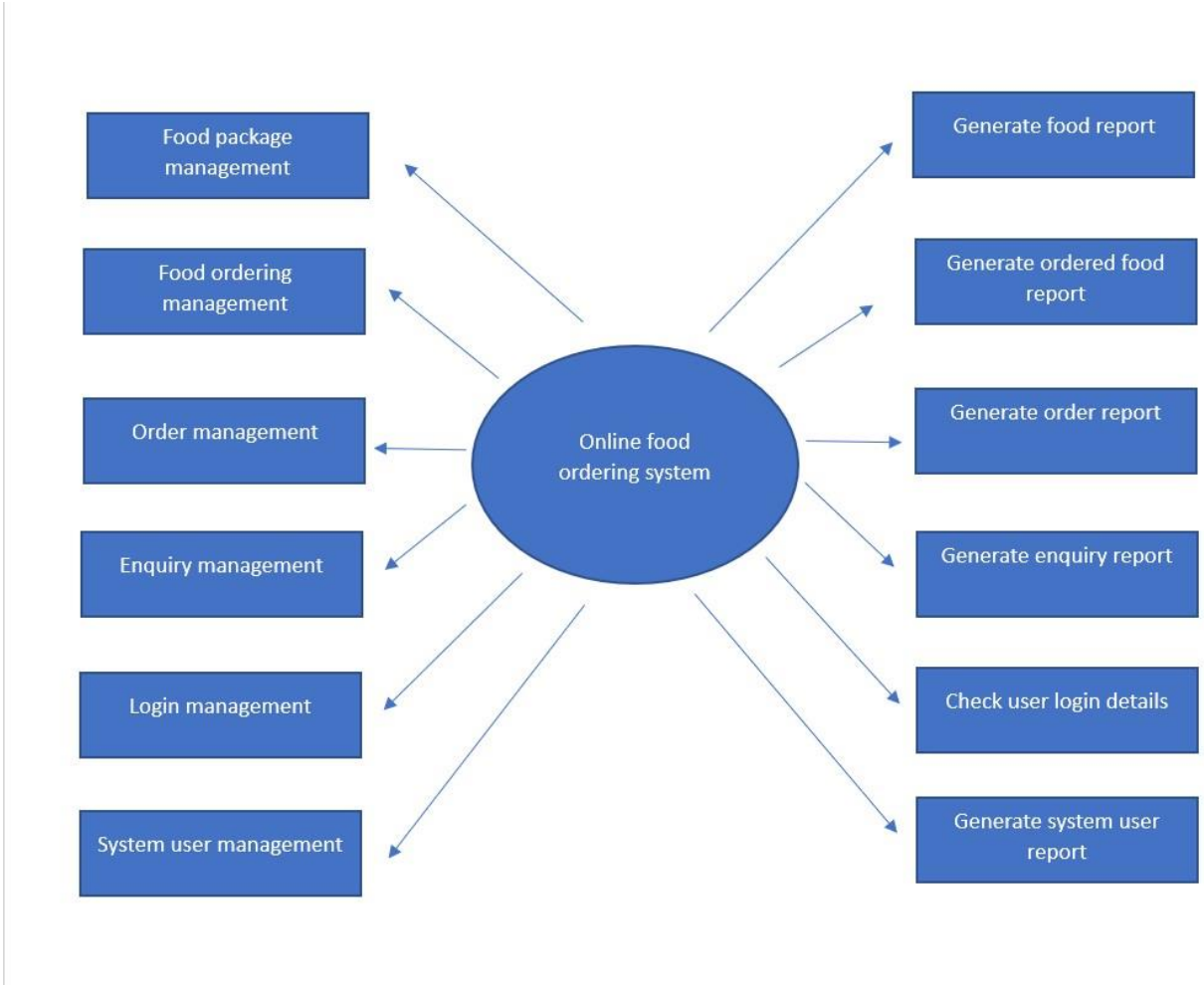

Figure 3.2 1st Level DFD diagram- online food ordering platform

We usually begin by drawing a context diagram, a simple representation of the whole system. Context level DFD, also known as level 0 DFD, sees the whole system as a single process and emphasis the interaction between the system and external entities.

To elaborate further from that, we drill down to a level 1 diagram with additional information about the major functions of the system. This could continue to evolve to become a level 2 diagram when further analysis is required.

The Level 1 DFD shows how the system is divided into sub-systems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole.

3.5.3 USE CASE DIAGRAM

Use case diagrams are considered for high level requirement analysis of a system Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified. Now when the initial task is complete use case diagrams are modeled to present the outside view.

Use case:

Use case diagrams are considered for high level requirement analysis of a system.So when the requirements of a system are analyzed the functionalities are captured in use cases. So we can say that uses cases are nothing but the system functionalities written in an organized manner.

Actor:

Now the second things which are relevant to the use cases are the actors. Actors can be defined as something that interacts with the system. The actors can be human user, some internal applications or may be some external applications.

Relationship:

Relationships exist among the use cases and actors. Show relationships and dependencies clearly in the diagram. Do not try to include all Because the main purpose of the diagram is to identify requirements. types of relationships.

ADMIN

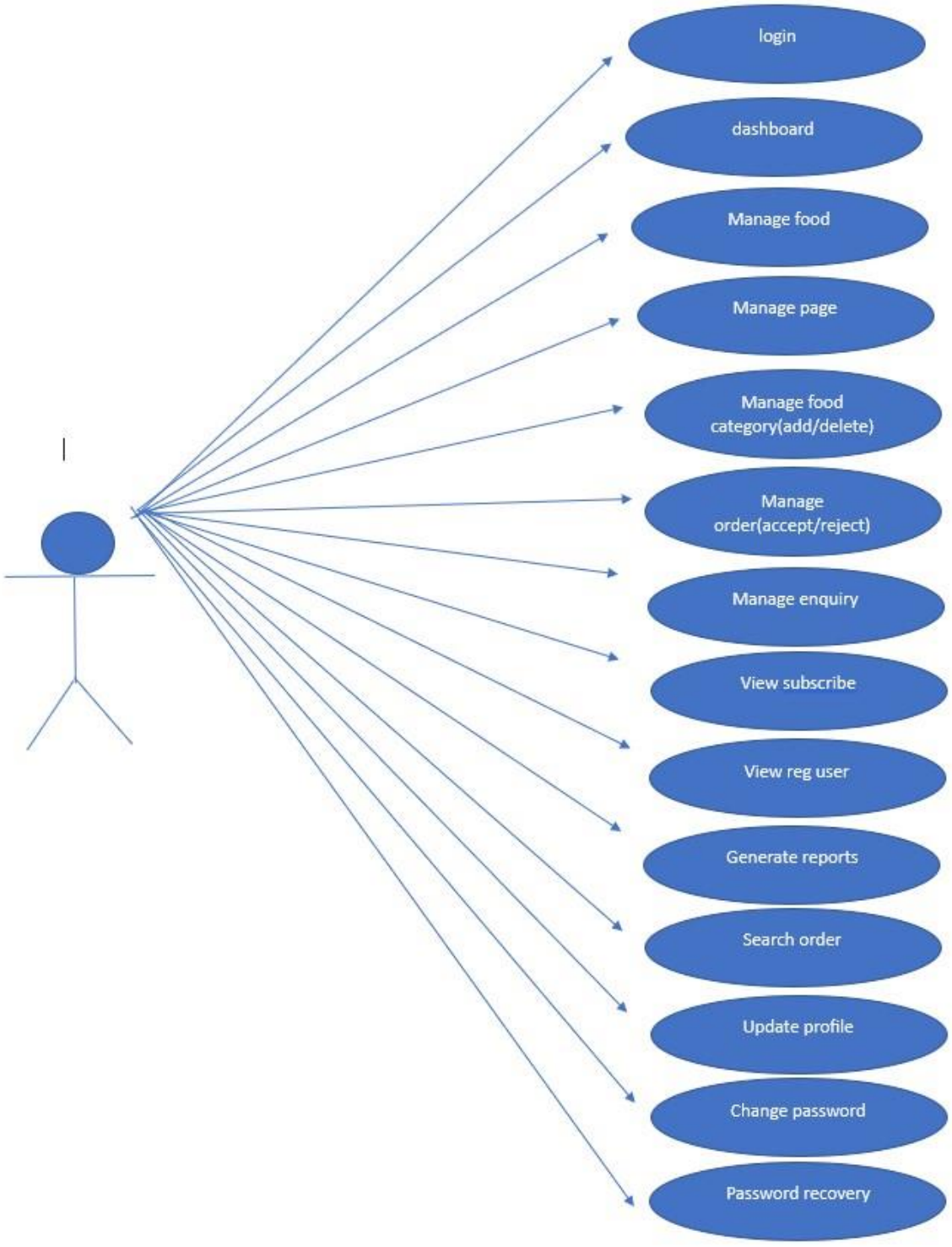

Figure 3.4 Features of a online food ordering platform

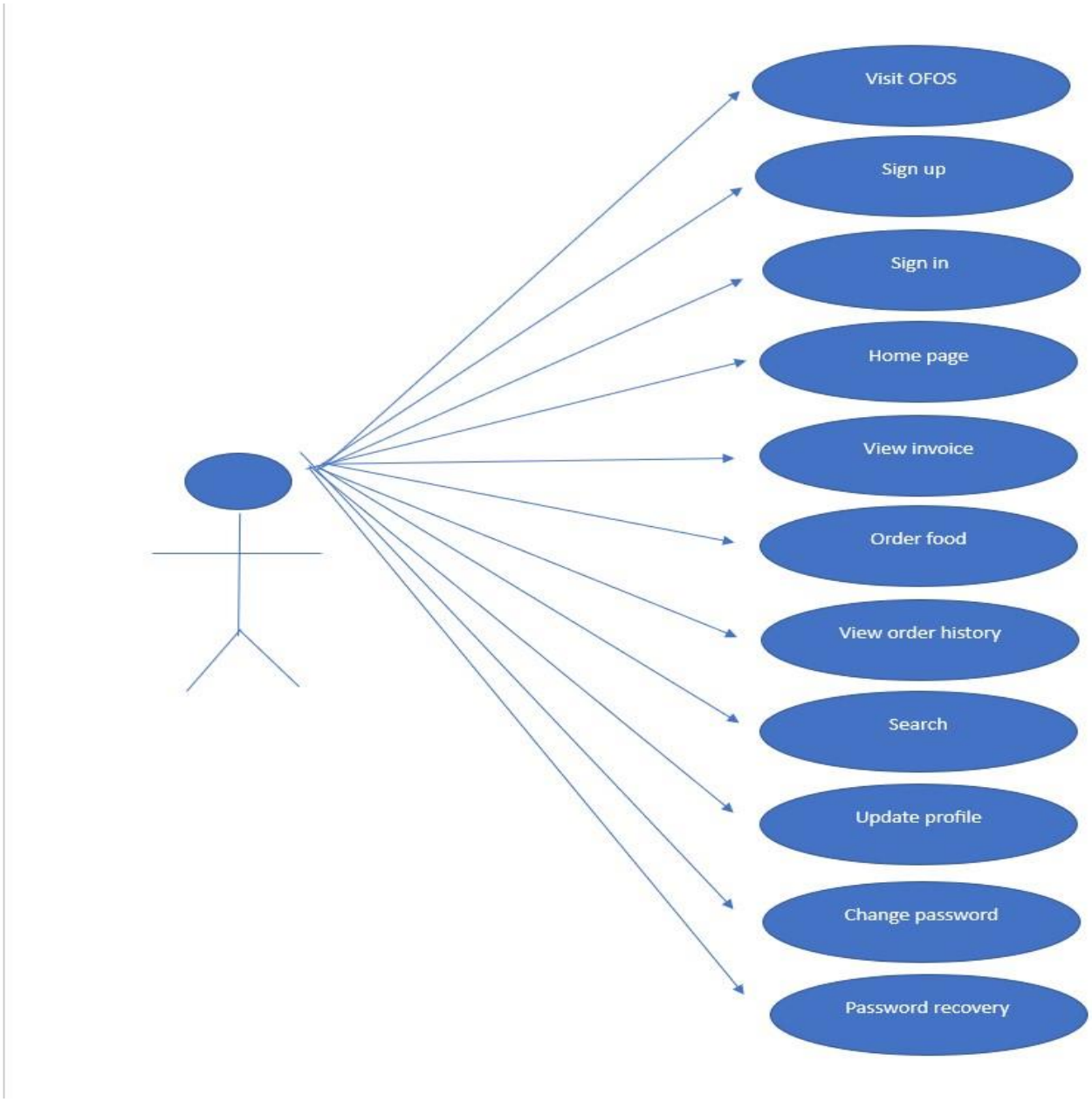

Figure 3.5 User-side functionalities of a food ordering platform

SUMMARY

This Chapter describes the Input design, Output design, Module Description, such as Admin module and User module and data model. In the Admin module, the process is add products and delivery. User module, add details and login. The Data flow diagram, class diagram. database design, data integrity process also discussed in this chapter. The system implementation for this project are described in Chapter 4.

.

CHAPTER 4

SYSTEM IMPLEMENTATION

4.1 SYSTEM IMPLEMENTATION

Implementation is the stage in the project where the theoretical design is turned into a working system. The most critical stage is achieving a successful system and giving confidence in the new system for the users, that it will work efficiently and effectively, It involves careful planning, investing of the current system, and its constraints on implementation, design of methods to achieve the changeover methods The implementation process begins with preparing a plan for the implementation of the system. According to this plan, the activities are to be carried out in these plans; discussion has been made regarding the equipment, resources, and how to test activities.

The coding step translates a detailed design representation into a programming language realization. Programming languages are vehicles for communication between humans and computers programming language characteristics and coding styles can profoundly affect software quality and maintainability. The coding is done with the following characteristics in

- Ease of design to code translation.
- Code efficiency.
- Memory efficiency
- Maintainability

The user should be very careful while implementing a project to ensure what they have planned is properly implemented. The user should not change the purpose of project while implementing The user should not go in a roundabout way to achieve a solution; it should be direct, crisp and clear and up to the point.

4.2 STANDARDIZATION OF CODING

ReactJS follows few rates and maintains its style of coding. Making me of indentation for indicating the start and end of control structures along with a clear specification of where the code is between them. The consistency in the naming convention of the variables is followed throughout the code. Also, the data should be described in the code Comment lines are used in this code it helps the developers to understand the code. The use of comment lines in this code aids in the understanding of the code by the developers. This code has been written to improve readability without affecting the code's essential functioning. This code is done to enhance the readability of the code without modifying the basic functionality of the code.

4.3 ERROR HANDLING

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program. Whenever an exception occurs, the program halts the execution, and thus the further code is not executed. Therefore, an exception is the error which ReactJS is unable to tackle with, ReactJS provides us with the way to handle the Exception so that the other part of the code can be executed without any disruption. ReactJS gives us a technique to handle the Exception so that the rest of the code may continue to run without interruption.

SUMMARY

In this chapter, Systems implementation is the process of defining how the information system should be built ensuring that the information system is operational and used, ensuring that the information system meets quality standard. we had gained knowledge on system implementation, and how to write a standardized code with command lines that a developer can understand and code readability and whenever exception occurs error can be handled. Testing and results for the project is described in chapter 5.

CHAPTER 5

TESTING AND RESULTS

5.1 SYSTEM TESTING

Software testing is a critical element of software quality assurance and represents the ultimate review of specifications, design, and coding. The testing phase involves the testing of the system using various test data. Preparation of test data plays a vital role in the system testing. After the preparation of the test data, the system under study is tested. After the source code has been completed, documented as related data structures. The completed project has to undergo testing and validation where there is the subtitle and definite attempt to get errors.

The project developer is always responsible for testing the individual units i.e. modules of the program. In many cases developer also conducts integration testing i.e. the testing step that leads to the construction of the complete program structure.

This project has undergone the following testing procedures to ensure its correctness

- Unit testing
- Integration Testing
- Validation Testing

5.2 UNIT TESTING

In the unit testing, the analyst tests the program making up a system. The software units in a system are the modules and routines that are assembled and integrated to perform a specific function. In a large system, many modules on different levels are needed. Unit testing can be performed from the bottom up starting with the smallest and lowest level modules and proceeding one at a time. For each module in a bottom-up testing. a short program executes the module and provides the needed data.

```
onCheckout: (userFormData: UserFormData) => void;
  disabled: boolean;
  isLoading: boolean;
};
const CheckoutButton = ({ onCheckout, disabled, isLoading }: Props) => {
  const {
    isAuthenticated,
    isLoading: isAuthLoading,
    loginWithRedirect,
  } = useAuth0();

  const { pathname } = useLocation();

  const { currentUser, isLoading: isGetUserLoading } = useGetMyUser();

  const onLogin = async () => {
    await loginWithRedirect({
```

```
    appState: {
      returnTo: pathname,
    },
  });
};
```

5.3 INTEGRATION TESTING

Integration testing is a systematic technique for constructing the program structure while conducting a test to uncover errors associated with interfacing. Objectives are used to take unit test modules and built a program structure that has been directed by design. The integration testing is performed for this Project when all the modules were to make it a complete system. After integration, the project works successfully.

5.4 VALIDATION TESTING

Validation testing can be defined in many ways, but a simple definition is that can be reasonably expected by the customer. After the validation test has been conducted. one of two possible conditions exists.

The functions or performance characteristics confirm to specification and are accepted. A deviation from the specification is uncovered and a deficiency list is created.

The proposed system under consideration has been tested by using validation testing and found to be working satisfactorily. For example, in this project validation testing is performed against the user module. This module is tested with the following valid and invalid inputs for the field email id.

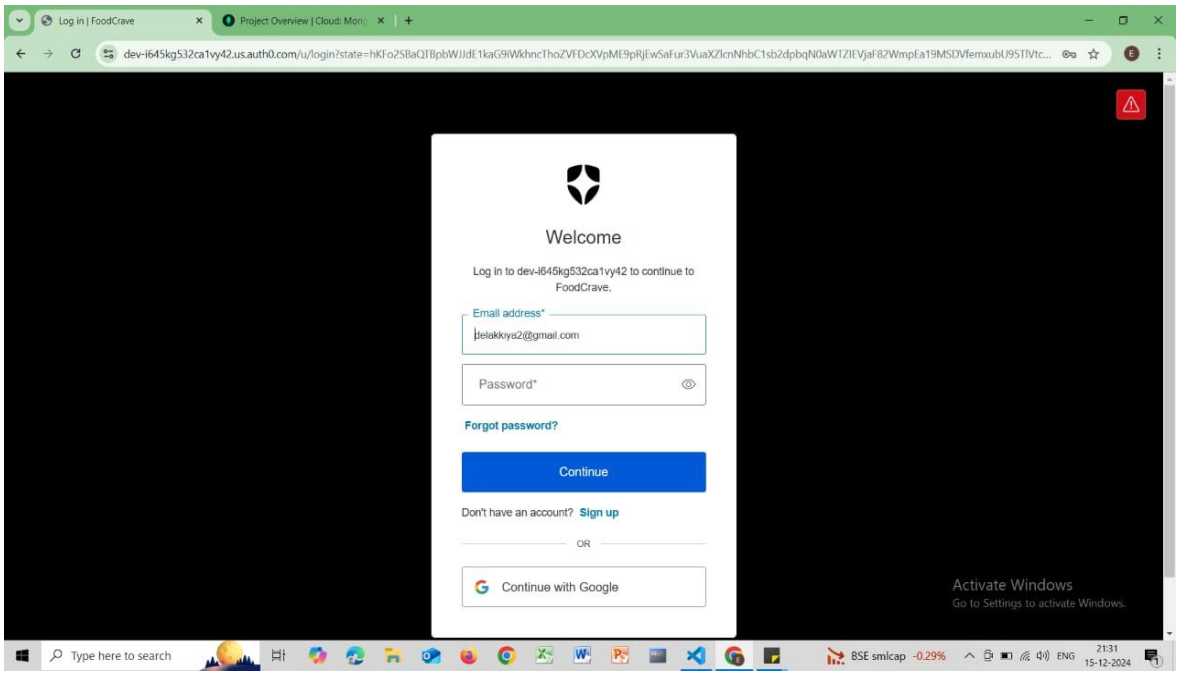

Figure 5.1 Signup page

In Figure 5.1, registration page which gives the error message that user already exist in the database. So, the user need to give new username or new email id to create account

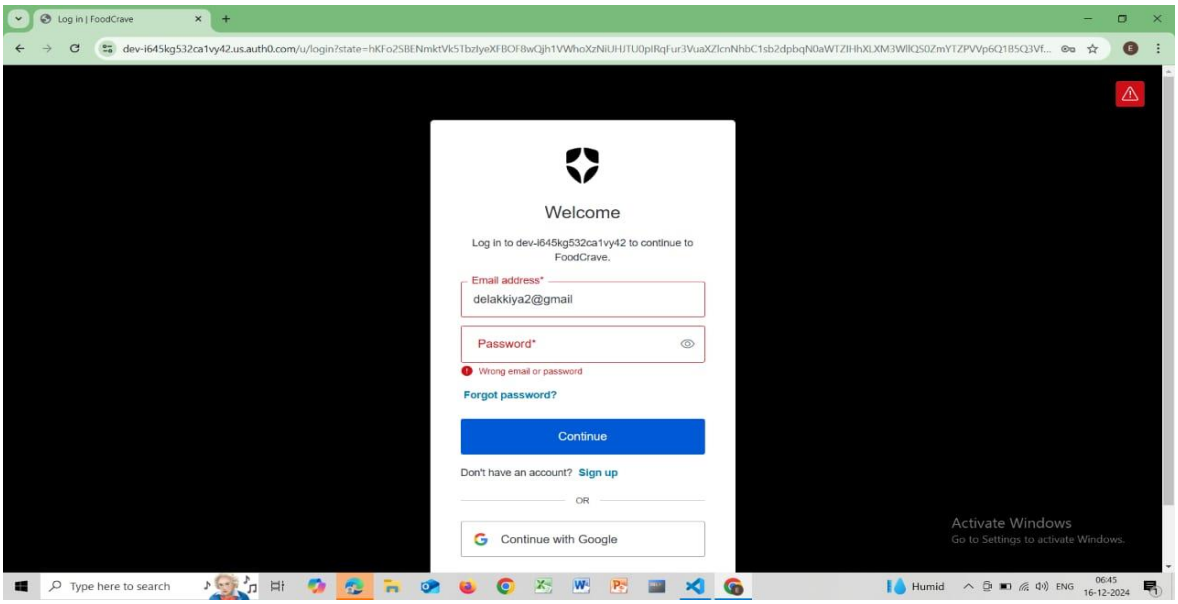

Figure 5.2 login page

In Figure 5.2, when the user try's to login with the email id which not used in creating account, it will show the error message as Email not found. User can login using the valid email.

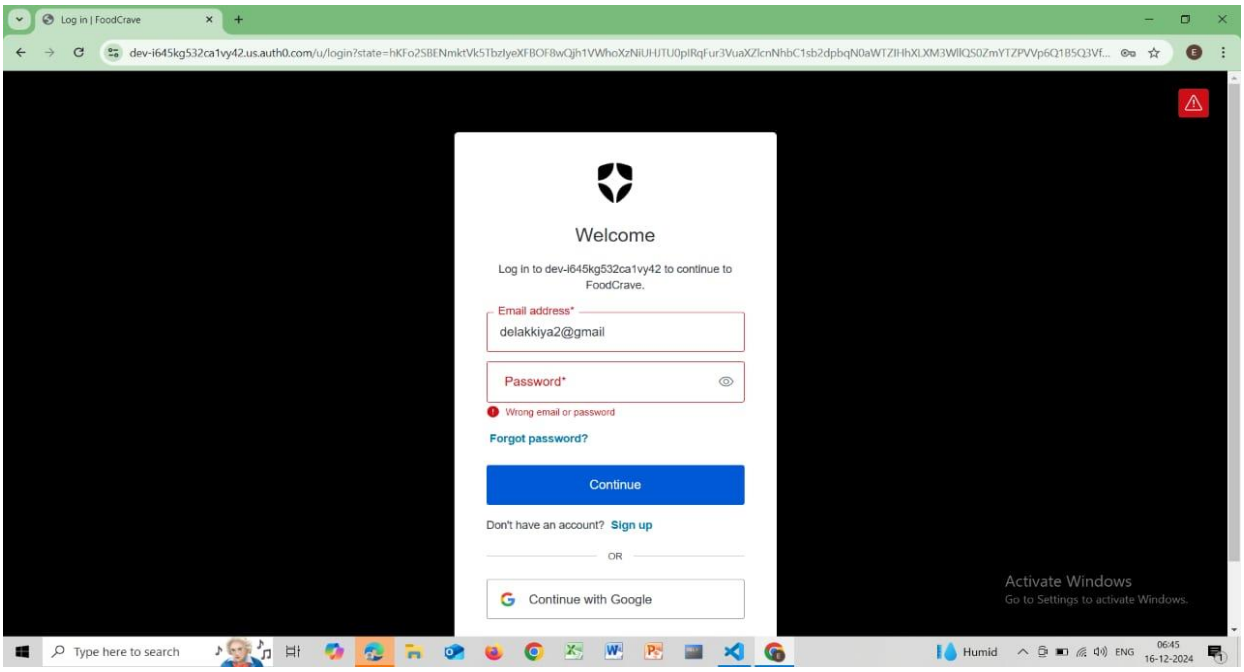

Figure 5.3 Login Page with Invalid password

In Figure 5.3, when the user try's to login with the wrong password, it will show the error message as Wrong password. User can login using the valid password

5.5 RESULTS

This issue includes the use of the application, the benefits of the admin, the result reached, and the technology utilized to achieve it. The way this application will work and the technologies that will be needed to make it work.

This project helped us in gaining valuable information and practical knowledge on several topics like designing web pages using ReactJS & NodeJs. Usage of responsive templates. Designing of android applications, and management of databases using firebase. The entire system is secured. Also, the project helped us understand the development phases of a project and software development life cycle. We learned how to test different features of a project.

Online shopping is a technology that will help business owners operate their day- to-day operations more smoothly and efficiently. The primary goal of this system was to suit the needs of the sales department. The detailed proposal for an online application for flowers discusses the applicability of an online shopping cart for flower in great depth. Furthermore, the app has a feature that allows users to create various types of reports that may be saved in the app. Before the product is sent to the customer, they have the option to change their address as well as the product. The user interface should be simple enough for the typical individual to grasp. This will be one of the most enjoyable applications to develop and implement in the real world.

SUMMARY

In this chapter, it shows the system testing and its results. System testing is the testing of a system as a whole. End to end testing is performed to verify that all the scenarios are working as expected. In result, The challenges experienced have been thoroughly outlined in this chapter, and all of those challenges have been overcome by achieving this online Food ordering platform. The conclusion and future work for the project is described in Chapter 6.

CHAPTER 6

CONCLUSION AND FUTURE ENHANCEMENTS

6.1 CONCLUSION

The project "ONLINE FOOD ORDERING PLATFORM has been designed and developed as per the specification. The project is very simple and gives clear understanding. The code written in this project is very clear. The system is tested with various sample data and produce correct result.

From a proper analysis of positive points and constraints on the component, it can be safely concluded that the product is a highly efficient GUI based component. This application is working properly and meeting to all user requirements. This application is up and running and meets all of the user's needs. This component can easily be integrated into a variety of other systems. This component can be easily plugged in many other systems. It's really user friendly application to user. It will reduce the time of user and effort also. In future, we have to change application to a web application there by enabling to adopt for branches.

The new system eliminates the difficulties in the existing system. It is developed in a user friendly manner. The system is very fast according to the user wish that can be viewed or retaken at any level.

6.2 FUTURE ENHANCEMENTS

Future enhancements for an Online Food Ordering Platform can significantly improve user experience and operational efficiency. Integrating AI could offer personalized food recommendations, dynamic pricing, and smart chatbots for better customer service. IoT integration could allow real-time tracking of deliveries and smart kitchen devices for better order management. Payment options could expand to include cryptocurrencies and biometric authentication for added security. Delivery methods could evolve with drone or autonomous vehicle options, and adding gamification or loyalty programs could increase user engagement.

APPENDIX 1

SAMPLE CODING

```
import { Request, Response } from "express";
import Restaurant from "../models/restaurant";
import cloudinary from "cloudinary";
import mongoose from "mongoose";
import Order from "../models/order";

const getMyRestaurant = async (req: Request, res: Response) => {
  try {
    const restaurant = await Restaurant.findOne({ user: req.userId });
    if (!restaurant) {
      return res.status(404).json({ message: "restaurant not found" });
    }
    res.json(restaurant);
  } catch (error) {
    console.log("error", error);
    res.status(500).json({ message: "Error fetching restaurant" });
  }
};
```

```
const createMyRestaurant = async (req: Request, res: Response) => {
  try {
    const existingRestaurant = await Restaurant.findOne({ user: req.body.userId });

    if (existingRestaurant) {
      return res
        .status(409)
        .json({ message: "User restaurant already exists" });
    }

    const imageUrl = await uploadImage(req.file as Express.Multer.File);

    const restaurant = new Restaurant(req.body);
    restaurant.imageUrl = imageUrl;
    restaurant.user = new mongoose.Types.ObjectId(req.body.userId);
    restaurant.lastUpdated = new Date();
    await restaurant.save();
    res.status(201).send(restaurant);
  } catch (error) {
    console.log(error);
    res.status(500).json({ message: "Something went wrong" });
  }
};
```

```
const updateMyRestaurant = async (req: Request, res: Response) => {
  try {
    const restaurant = await Restaurant.findOne({
      user: req.userId,
    });
    if (!restaurant) {
      return res.status(404).json({ message: "restaurant not found" });
    }
    restaurant.restaurantName = req.body.restaurantName;
    restaurant.city = req.body.city;
    restaurant.country = req.body.country;
    restaurant.deliveryPrice = req.body.deliveryPrice;
    restaurant.estimatedDeliveryTime = req.body.estimatedDeliveryTime;
    restaurant.cuisines = req.body.cuisines;
    restaurant.menuItems = req.body.menuItems;
    restaurant.lastUpdated = new Date();

    if (req.file) {
      const imageUrl = await uploadImage(req.file as Express.Multer.File);
      restaurant.imageUrl = imageUrl;
    }
```

```
    await restaurant.save();
    res.status(200).send(restaurant);
  } catch (error) {
    console.log("error", error);
    res.status(500).json({ message: "Something went wrong" });
  }
};
const getMyRestaurantOrders = async (req: Request, res: Response) => {
  try {
    const restaurant = await Restaurant.findOne({ user: req.userId });
    if (!restaurant) {
      return res.status(404).json({ message: "restaurant not found" });
    }
    const orders = await Order.find({ restaurant: restaurant._id })
     .populate("restaurant")
    .populate("user");
    res.json(orders);
  } catch (error) {
    console.log(error);
    res.status(500).json({ message: "something went wrong" });
  }
};
const updateOrderStatus = async (req: Request, res: Response) => {
```

```
  try {
    const { orderId } = req.params;
    const { status } = req.body;

    const order = await Order.findById(orderId);
    if (!order) {
      return res.status(404).json({ message: "order not found" });
    }
    const restaurant = await Restaurant.findById(order.restaurant);
    if (restaurant?.user?._id.toString() !== req.userId) {
      return res.status(401).send();
    }
    order.status = status;
    await order.save();
    res.status(200).json(order);
  } catch (error) {
    console.log(error);
    res.status(500).json({ message: "unable to update order status" });
  }
};

const uploadImage = async (file: Express.Multer.File) => {
  const image = file;
```

```
  const base64Image = Buffer.from(image.buffer).toString("base64");
  const dataURI = data:${image.mimetype};base64,${base64Image};

  const uploadResponse = await cloudinary.v2.uploader.upload(dataURI);
  return uploadResponse.url;
};
export default {
  updateOrderStatus,
  getMyRestaurantOrders,
  getMyRestaurant,
  createMyRestaurant,
  updateMyRestaurant,
};
import { Request, Response } from "express";
import User from "../models/user";
const getCurrentUser = async (req: Request, res: Response) => {
  try {
    const currentUser = await User.findOne({ _id: req.userId });
    if (!currentUser) {
      return res.status(404).json({ message: "User not found" });
    }
    res.json(currentUser);
  } catch (error) {
```

```
    console.log(error);
    return res.status(500).json({ message: "Something went wrong" });
  }
};
const createCurrentUser = async (req: Request, res: Response) => {
  try {
    const { auth0Id } = req.body;
    const existingUser = await User.findOne({ auth0Id });
    if (existingUser) {
      return res.status(200).send();
    }
    const newUser = new User(req.body);
    await newUser.save();
    res.status(201).json(newUser.toObject());
  } catch (error) {
    console.log(error);
    res.status(500).json({ message: "Error creating user" });
  }
};
const updateCurrentUser = async (req: Request, res: Response) => {
  try {
    const { name, addressLine1, country, city } = req.body;
    const user = await User.findById(req.userId);
```

```
    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
    user.name = name;
    user.addressLine1 = addressLine1;
    user.city = city;
    user.country = country;
    await user.save();
    res.send(user);
  } catch (error) {
    console.log(error);
    res.status(500).json({ message: "Error updating user" });
  }
};
export default {
  getCurrentUser,
  createCurrentUser,
  updateCurrentUser,
};
import Stripe from "stripe";
import { Request, Response } from "express";
import Restaurant, { MenuItemType } from "../models/restaurant";
import Order from "../models/order";
```

```
const STRIPE = new Stripe(process.env.STRIPE_API_KEY as string);
const FRONTEND_URL = process.env.FRONTEND_URL as string;
const STRIPE_ENDPOINT_SECRET = process.env.STRIPE_WEBHOOK_SECRET as string;


const getMyOrders = async (req: Request, res: Response) => {
  try {
    const orders = await Order.find({ user: req.userId })
    .populate("restaurant")
    .populate("user");
    res.json(orders);
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: "Something went wrong" });
  }
};
type CheckoutSessionRequest = {
  cartItems: {
    menuItemId: string;
    name: string;
    quantity: string;
  }[];
  deliveryDetails: {
    email: string;
```

```
    name: string;
    addressLine1: string;
    city: string;
  };
  restaurantId: string;
};
const stripeWebhookHandler = async (req: Request, res: Response) => {
  let event;
  try {
    const sig = req.headers["stripe-signature"];
    event = STRIPE.webhooks.constructEvent(
      req.body,
      sig as string,
      STRIPE_ENDPOINT_SECRET
    );
  } catch (error: any) {
    console.error(error);
    return res.status(400).send(Webhook error: ${error.message});
  }
  if (event.type === "checkout.session.completed") {
    const order = await Order.findById(event.data.object.metadata?.orderId);
    if (!order) {
      return res.status(404).json({ message: "Order not found" });
```

```
    }
    order.totalAmount = event.data.object.amount_total;
    order.status = "paid";

    await order.save();
  }

  res.status(200).send();
};
const createCheckoutSession = async (req: Request, res: Response) => {
  try {
    const checkoutSessionRequest: CheckoutSessionRequest = req.body;
    const restaurant = await Restaurant.findById(
      checkoutSessionRequest.restaurantId
    );
    if (!restaurant) {
      throw new Error("Restaurant not found");
    }
    const newOrder = new Order({
      restaurant: restaurant,
      user: req.userId,
      status: "placed",
      deliveryDetails: checkoutSessionRequest.deliveryDetails,
```

```
    cartItems: checkoutSessionRequest.cartItems,
    createdAt: new Date(),
  });

  const lineItems = createLineItems(
    checkoutSessionRequest,
    restaurant.menuItems
  );
  const session = await createSession(
    lineItems,
    newOrder._id.toString(),
    restaurant.deliveryPrice,
    restaurant._id.toString(),
    checkoutSessionRequest.deliveryDetails
  );
  if (!session.url) {
    return res.status(500).json({ message: "Error creating Stripe session" });
  }
  await newOrder.save();
  res.json({ url: session.url });
} catch (error: any) {
  console.error(error);
  res.status(500).json({ message: error.message });
```

```
  }
};

const createLineItems = (
  checkoutSessionRequest: CheckoutSessionRequest,
  menuItems: MenuItemType[]
) => {
  const lineItems = checkoutSessionRequest.cartItems.map((cartItem) => {
    const menuItem = menuItems.find(
      (item) => item._id.toString() === cartItem.menuItemId.toString()
    );
    if (!menuItem) {
      throw new Error(Menu item not found: ${cartItem.menuItemId});
    }
    const line_item: Stripe.Checkout.SessionCreateParams.LineItem = {
      price_data: {
        currency: "inr",
        unit_amount: menuItem.price,
        product_data: {
          name: menuItem.name,
        },
      },
      quantity: parseInt(cartItem.quantity),
```

```
    };
    return line_item;
  });

  return lineItems;
};
const createSession = async (
  lineItems: Stripe.Checkout.SessionCreateParams.LineItem[],
  orderId: string,
  deliveryPrice: number,
  restaurantId: string,
  deliveryDetails: {
    email: string;
    name: string;
    addressLine1: string;
    city: string;
  }
) => {
  const sessionData = await STRIPE.checkout.sessions.create({
    line_items: lineItems,
    shipping_options: [
      {
        shipping_rate_data: {
```

```
        display_name: "Delivery",
        type: "fixed_amount",
        fixed_amount: {
          amount: deliveryPrice,
          currency: "inr",
        },
      },
    },
  ],
  mode: "payment",
  metadata: {
    orderId,
    restaurantId,
  },
  customer_email: deliveryDetails.email,
  shipping_address_collection: {
    allowed_countries: ["IN"],
  },
  success_url: ${FRONTEND_URL}/,
  cancel_url: ${FRONTEND_URL}/detail/${restaurantId}?cancelled=true,
});

return sessionData;
```

```
};



export default {
  getMyOrders,
  createCheckoutSession,
  stripeWebhookHandler,
};
import express from "express";
import { param, query } from "express-validator";
import RestaurantController from "../controllers/RestaurantController";
const router = express.Router();
router.get(
  "/nearby",
  query("maxDeliveryTime")
    .isNumeric()
    .withMessage("maxDeliveryTime query parameter must be a valid number")
    .notEmpty()
    .withMessage("maxDeliveryTime query parameter is required"),
  query("city")
    .isString()
    .withMessage("city query parameter must be a valid string")
    .notEmpty()
```

```
    .withMessage("city query parameter is required"),
  RestaurantController.getNearbyRestaurants
);

router.get(
  "/:restaurantId",
  param("restaurantId")
    .isString()
    .trim()
    .notEmpty()
    .withMessage("RestaurantId paramenter must be a valid string"),
  RestaurantController.getRestaurant
);
router.get(
  "/search/:city",
  param("city")
    .isString()
    .trim()
    .notEmpty()
    .withMessage("City paramenter must be a valid string"),
  RestaurantController.searchRestaurant
);
export default router;
```

```
import { useAuth0 } from "@auth0/auth0-react";
import { useLocation } from "react-router-dom";
import { Button } from "./ui/button";
import LoadingButton from "./LoadingButton";
import { Dialog, DialogContent, DialogTrigger } from "./ui/dialog";
import UserProfileForm, {
  UserFormData,
} from "@/forms/user-profile-form/UserProfileForm";
import { useGetMyUser } from "@/api/MyUserApi";
type Props = {
  onCheckout: (userFormData: UserFormData) => void;
  disabled: boolean;
  isLoading: boolean;
};
const CheckoutButton = ({ onCheckout, disabled, isLoading }: Props) => {
  const {
    isAuthenticated,
    isLoading: isAuthLoading,
    loginWithRedirect,
  } = useAuth0();

  const { pathname } = useLocation();
```

```
const { currentUser, isLoading: isGetUserLoading } = useGetMyUser();


const onLogin = async () => {
  await loginWithRedirect({
    appState: {
      returnTo: pathname,
    },
  });
};
if (!isAuthenticated) {
  return (
    <Button onClick={onLogin} className="bg-orange-500 flex-1">
      Log in to check out
    </Button>
  );
}
if (isAuthLoading || !currentUser || isLoading) {
  return <LoadingButton />;
}
return (
  <Dialog>
    <DialogTrigger asChild>
      <Button disabled={disabled} className="bg-orange-500 flex-1">
```

```
          Go to checkout
        </Button>
      </DialogTrigger>
      <DialogContent className="max-w-[425px] md:min-w-[700px] bg-gray-50">
        <UserProfileForm
          currentUser={currentUser}
          onSave={onCheckout}
          isLoading={isGetUserLoading}
          title="Confirm Deliery Details"
          buttonText="Continue to payment"
        />
      </DialogContent>
    </Dialog>
  );
};
export default CheckoutButton;
```

APPENDIX 2

SCREENSHOTS

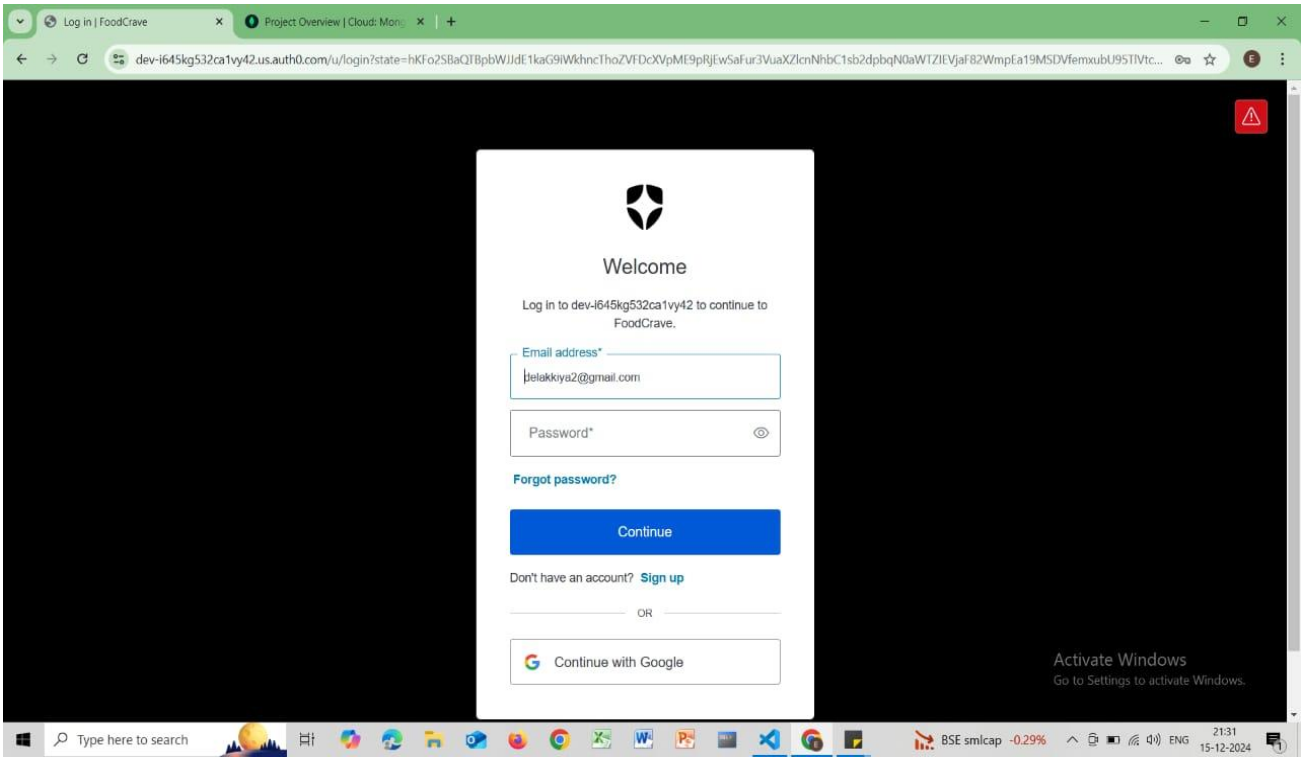

Figure A 2.1 Signup page for user- food ordering platform

In Figure A 2.1, It features a clean interface with fields for entering an email address and password, along with options for password recovery and signing up for a new account. Users can log in by clicking the "Continue" button after entering their credentials or choose to sign in with Google for convenience. The design is user-friendly, guiding individuals through the authentication process efficiently, whether they are returning users or new to the platform.

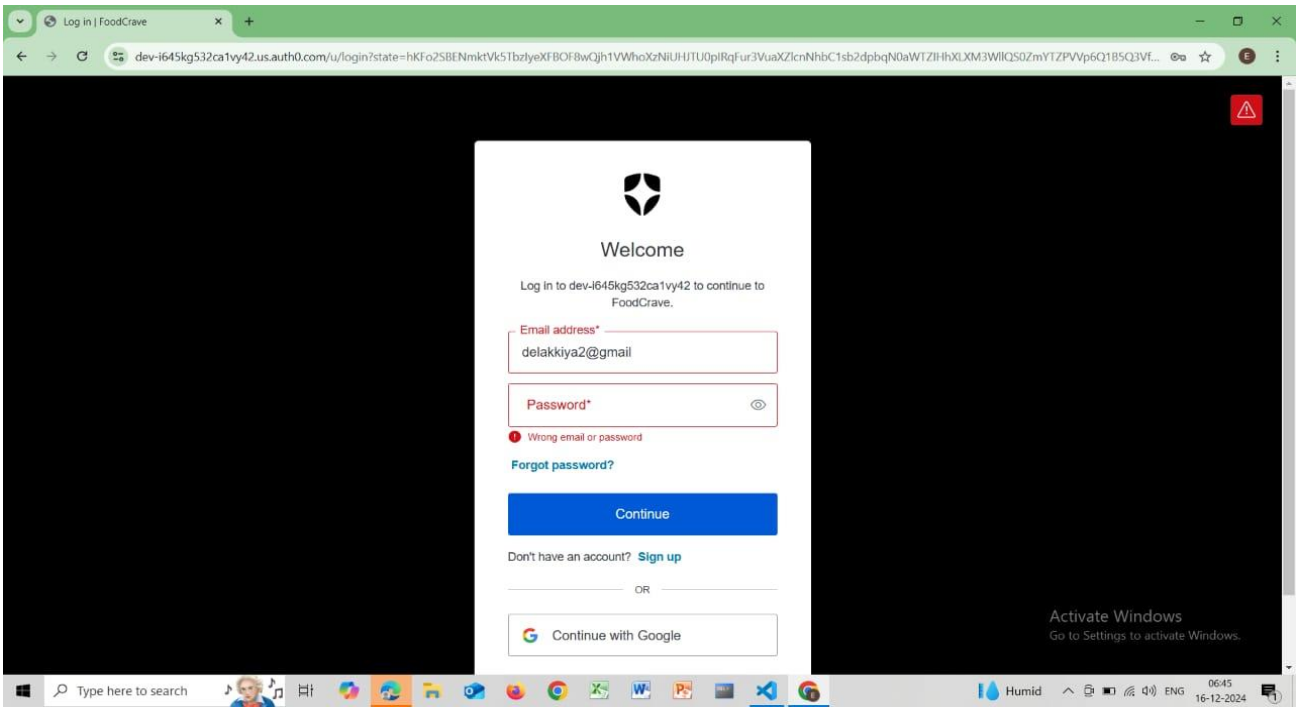

Figure A 2.2 Incorrect Credentials

In Figure A 2.2, the user enters their email address and password to log in but encounters an error message stating "Wrong email or password," indicating incorrect credentials.

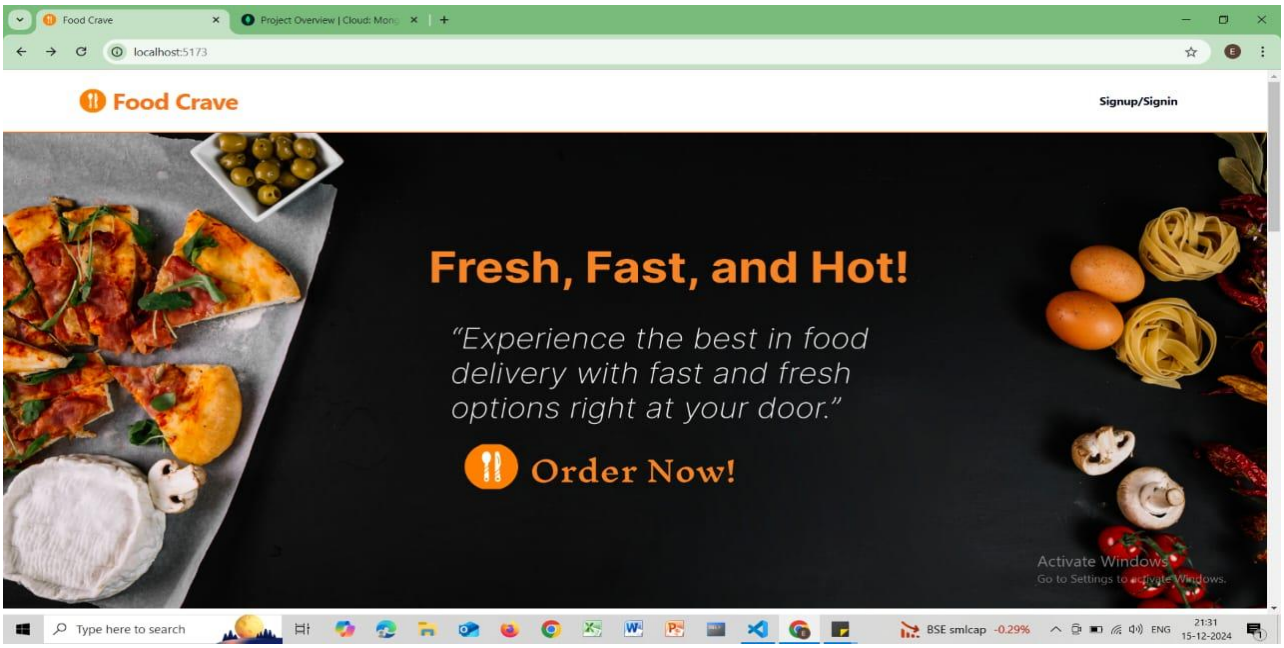

Figure A 2.3 Home page-food ordering platform

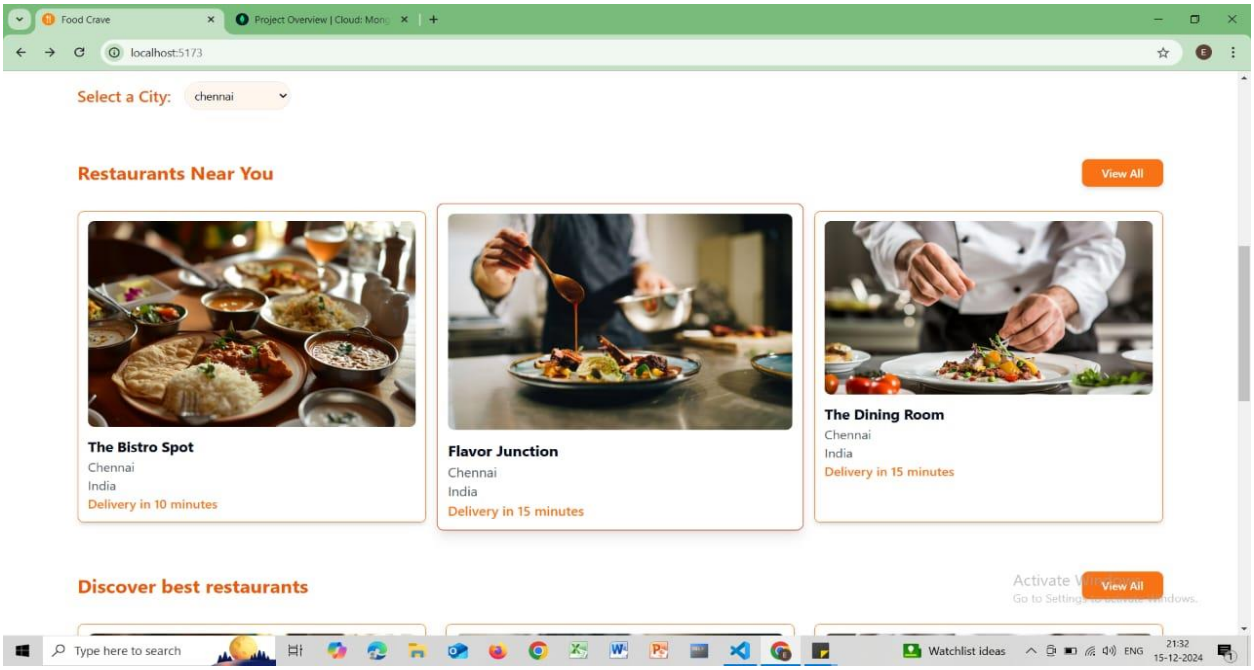

Figure A 2.4 Search the Restaurants pages-food ordering platform

In Figure A2.4, The featured restaurants include "The Bistro Spot," "Flavor Junction," and "The Dining Room," all located in Chennai, India, with delivery times mentioned (e.g., "Delivery in 10 minutes"). A "View All" button is also present, allowing users to explore additional options.

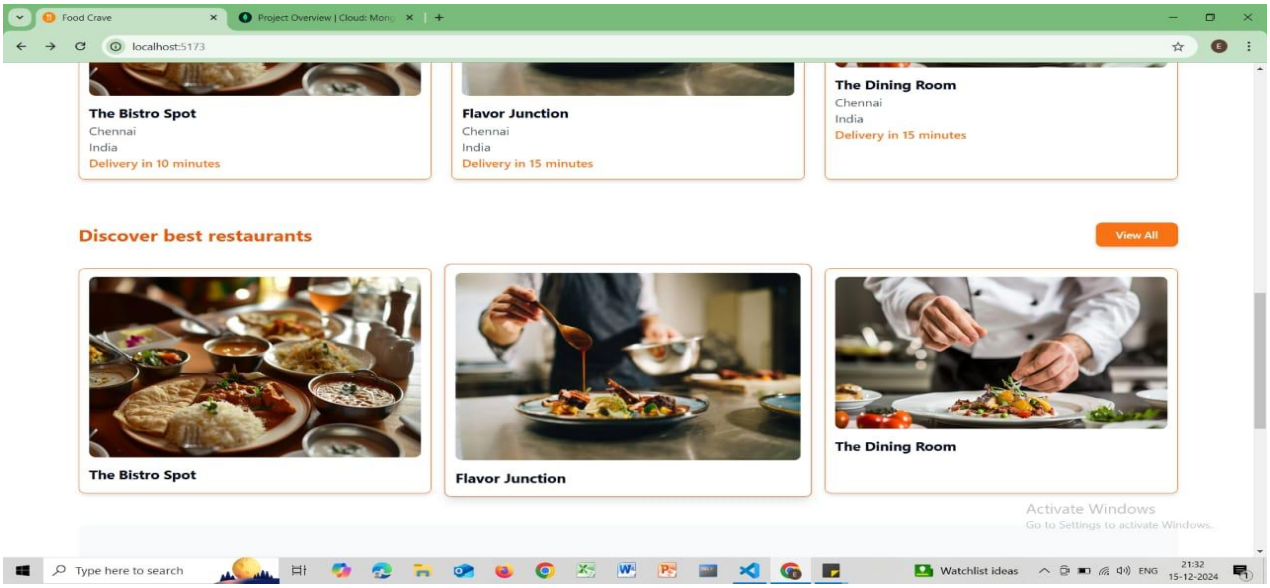

Figure A 2.5 View recipes page- food ordering platform

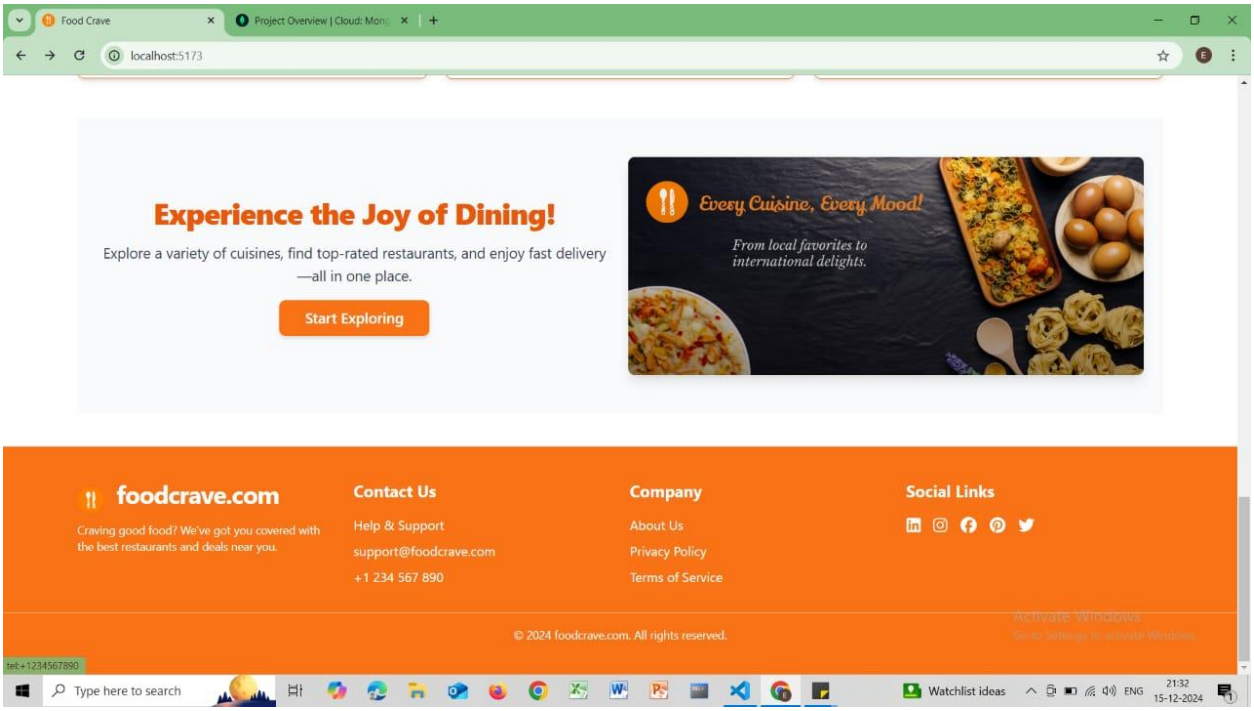

Figure A 2.6 Explore My Page- food ordering platform

In Figure A2.6, The users can explore the Food Crave website for support, updates, and deals through features like live chat, social media links, and essential company details. It invites visitors to stay connected and visit again for a seamless experience.

REFERENCES

BOOKS:

[1]Rajesh Kumar,"*Online Food Ordering Systems: Design and Development"*,TechInsights Publishing, 2nd Edition,2021.

[2] Sarah Johnson, "*E-Commerce and Food Delivery Platforms: Innovations and Challenges"*,Digital World Press, 1st Edition,2022

[3] Michael Evans," *The Future of Food Delivery: Market Trends and Technology"*, Modern Business Publications,3rd Edition,2020

[4]Priya Sharma, "*Building Food Ordering Platforms: A Guide for Entrepreneurs"*,StartUp Guides, 1st Edition,2019

[5]Martin Kleppmann, "Designing Data-Intensive Applications", O'Reilly Media, 1st Edition,2017

[6]Don Norman,"The Design of Everyday Things", Basic Books, Revised and Expanded Edition, 2013

[7] Jeff Gothelf, **"**Designing Great Products with Agile Teams", O'Reilly Media, 2nd Edition, 2013

WEBSITES:

Behance (Food Ordering UI/UX Designs) – https://www.behance.net/search/projects?search=food%20ordering

Envato Elements (Food Ordering Website Templates) –

https://elements.envato.com

Dribbble (Food Ordering UI Designs) –

https://dribbble.com/search/food%20ordering