VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JNANASANGAMA, BELAGAVI - 590018

DBMS Project Report (21CSE143)

on

GoFood – A Food Ordering App

Submitted in partial fulfillment for the award of degree of

Bachelor of Engineering
in
COMPUTER SCIENCE AND ENGINEERING

Submitted by

1BG21CS001 Abhiram B S
1BG21CS003 Aditya B Prahalad

Guide
Dr. Amith Shekhar C
Associate Professor, Dept. of CSE
BNMIT, Bengaluru

Vidyayāmruthamashnuthe

B.N.M. Institute of Technology

An Autonomous Institution under VTU
Approved by AICTE, Accredited as grade A Institution by NAAC. All eligible branches – CSE, ECE, EEE, ISE & Mech. Engg. are Accredited by NBA for academic years 2018-19 to 2024-25 & valid upto 30.06.2025
URL: www.bnmit.org

Department of Computer Science and Engineering

2022 - 2023

B.N.M. Institute of Technology

An Autonomous Institution under VTU

Approved by AICTE, Accredited as grade A Institution by NAAC. All eligible branches – CSE, ECE, EEE, ISE & Mech. Engg. are Accredited by NBA for academic years 2018-19 to 2024-25 & valid upto 30.06.2025

URL: www.bnmit.org

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vidyayāmruthamashnuthe

CERTIFICATE

Certified that the Database Management System (21CSE143) project entitled **GoFood – A Food Ordering App** carried out by **Mr. Abhiram B S (1BG21CS001)**, **Mr. Aditya B Prahalad (1BG21CS003)**, are bonafide students of IV Semester, **BNM Institute of Technology** in partial fulfillment for the award of Bachelor of Engineering in COMPUTER SCIENCE AND ENGINEERING of **Visvesvaraya Technological University, Belagavi** during the year 2022-23. It is certified that all corrections / suggestions indicated for Internal Assessment have been incorporated in the project report.

Dr. Amith Shekhar C
Associate Professor
Department of CS
BNMIT, Bengaluru

ACKNOWLEDGEMENT

We would like to place on record our sincere thanks and gratitude to the concerned people, whose suggestions and words of encouragement has been valuable.

We express our heartfelt gratitude to the management of **BNM Institute of Technology**, for givingus the opportunity to pursue Degree of Computer Science and Engineering and helping us to shape our career. We take this opportunity to thank **Shri. Narayan Rao R. Maanay**, Secretary, **Prof. T. J. Rama Murthy**, Director, **Dr. S. Y. Kulkarni**, Additional Director, **Prof. Eishwar N Maanay**, Dean and **Dr. Krishnamurthy G. N.**, Principal for their support and encouragement to pursue this project. We would like to thank **Dr. Chayadevi M L**, Professor and Head, Dept. of Computer Science and Engineering, for her support and encouragement.

We would like to thank our guide **Dr. Amith Shekhar C**, Associate Professor, Dept. of Computer Science and Engineering, who has been the source of inspiration throughout our project work and has provided us with useful information at every stage of our project.

Finally, we are thankful to all the teaching and non-teaching staff of Department of Computer Science and Engineering for their help in the successful completion of our project. Last but not the least we would like to extend our sincere gratitude to our parents and all our friends who were a constant source of inspiration.

1BG21CS001 Abhiram B S
1BG21CS003 Aditya B Prahalad

ABSTRACT

In today's fast-paced world, the culinary industry is rapidly evolving, with a growing demand for convenient and efficient food ordering solutions. The food ordering web app addresses the common inconvenience of waiting in long queues at restaurants by providing a streamlined solution that benefits both customers and restaurant establishments.

GoFood – A Food Ordering App introduces a dynamic and innovative solution in the form of a food ordering web application developed using the MERN (MongoDB, Express.js, React, Node.js) stack. With the primary objective of enhancing user convenience and transforming the traditional dining experience, the app focuses on streamlining the ordering process and eliminating the need to stand in queues at restaurants.

Users of the app are presented with an intuitive and visually appealing interface. The React-based front-end offers a responsive and engaging design that facilitates easy navigation of menu items, and customization choices. Through this interface, customers can effortlessly browse a diverse range of dining establishments, explore detailed menus with enticing imagery, and personalize their orders according to their preferences and dietary requirements.

The core innovation of the app lies in its ability to empower customers to place their orders remotely, thus eliminating the need to physically queue at the restaurant. This not only reduces wait times but also provides users with the flexibility to customize the food based on their preference.

From a technical perspective, the back-end powered by Express.js and Node.js facilitates seamless communication between the user interface and the underlying database. MongoDB serves as the repository for storing user profiles, menu details, and order histories. This architecture ensures efficient data management, quick retrieval of information, and accurate order processing.

In conclusion, the MERN stack food ordering web app presents an innovative solution to enhance the dining experience by eliminating the need to stand in queues at restaurants. By blending cutting-edge technology with culinary experiences, this project offers a valuable solution that transforms the way individuals interact with food establishments. Through its user-friendly interface, efficient order management system, the app brings together the convenience of digital technology and the joy of culinary exploration. The synergy between user-friendly design, real-time communication, and streamlined ordering exemplifies the potential of technology to revolutionize the dining landscape.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER-I

Introduction

In an era defined by technological innovation and rapid lifestyle changes, traditional practices within the hospitality sector are evolving to meet the ever-growing expectations of modern consumers. One such challenge that has persisted is the inconvenience of waiting in queues for ordering food at hotels and restaurants.

Figure 1.1: Examples of Queue Systems in Hotels and Restaurants

Presenting an ingenious solution that seeks to reimagine and elevate the entire dining experience through the introduction of "GoFood – A Food Ordering App." This cutting-edge web application is strategically designed to empower customers with a streamlined, efficient, and user-friendly platform for placing food orders. The primary ambition is twofold: to provide patrons with a seamless and convenient avenue for requesting their desired meals while simultaneously addressing the pervasive inconvenience of enduring long queues at dining establishments. Through the integration of state-of-the-art technologies and an unwavering commitment to user-centric design principles, the food ordering web app not only simplifies the process of placing orders but also redefines the dining experience itself. It abolishes the necessity for physical waiting at eateries, promising a genuinely delightful and hassle-free interaction for every customer.

At the core of this endeavor lies the effective utilization of a robust Database Management System (DBMS). This system plays a pivotal role in ensuring a smooth and efficient user experience by adeptly managing critical data related to menus, orders, and customer profiles.

The project's architecture is built around a MongoDB Atlas cluster, ensuring secure and scalable storage for user data. The login and sign-up pages provide robust authentication mechanisms, safeguarding user information. The application's frontend is built using React framework and Bootstrap, offering an interactive and user-friendly experience.

By combining elements of web development, user interface design, Database Management System and digital innovation, this project seeks to bridge the gap between traditional dining practices and the convenience of technology, revolutionizing the way people interact with their dining choices. Through this project, this Food Ordering App project holds the promise of transforming the way patrons interact with dining establishments, eliminating the frustrations of queue-based systems.

Figure 1.2: Online Food Ordering Solution to address the queue system followed in Hotels and Restaurants

CHAPTER-II

Problem Statement

2.1 Statement of the problem

In the context of a swiftly evolving technological landscape and dynamic shifts in lifestyles, traditional norms within the hospitality sector are undergoing adaptations to meet the ever-heightening expectations of contemporary consumers. A persistent challenge within this evolving domain is the inconvenience encountered by patrons while queuing for food orders at hotels and restaurants. The process of standing in line not only consumes valuable time but also dampens the overall dining experience.

Despite advancements in various sectors, the hospitality industry continues to grapple with the issue of prolonged waiting times for food orders, a problem that dampens the overall dining experience. While digital solutions have made headway in enhancing various aspects of the hospitality experience, the process of ordering food has often been neglected, resulting in an imbalance between technological innovation and the conventional dining process. The lack of a user-centric approach in this area further exacerbates the problem, as current systems fail to provide an efficient means of food order placement and management. This results in lost time for customers, inefficiencies for restaurants, and a decreased overall level of customer satisfaction.

Addressing all issues the problem statement seeks to develop an innovative and user-friendly food ordering web application that not only simplifies the process of placing orders but also introduces a feature that eradicates the waiting-in-line dilemma, ensuring that customers can easily order food can enjoy their meals without the frustration of queuing up.

The introduction of "GoFood – A Food Ordering App" signifies a novel approach to addressing the problem of queue-based food ordering systems. The project not only aims to eliminate the inconvenience of queuing but also reimagines the entire dining experience. Through a combination of technological innovation, user-centric design, and efficient data management, this app aspires to revolutionize the way individuals interact with dining establishments, promising a seamless, efficient, and enjoyable food ordering process.

2.2 Objectives

The primary goals of this project involve creating a creative and user-centric web application for food ordering. This platform aims not only to streamline the order placement process but also to introduce a unique feature that eliminates the challenge of waiting in lines. This ensures that customers can effortlessly order their food and relish their meals without the inconvenience of queuing up. The objectives of the proposed solution, "GoFood – A Food Ordering App," are as follows:

- **Efficient Food Ordering Process:** Develop a streamlined and intuitive food ordering process that allows customers to easily browse menus, select items, customize preferences, and place orders seamlessly through the app.
- **Elimination of Queuing:** Implement a ground breaking feature that eradicates the need for physical queuing at hotels and restaurants. The app should ensure that customers can place their orders remotely, eliminating the frustration of waiting in lines.
- **User-Friendly Interface:** Design a user interface that is intuitive, visually appealing, and easy to navigate. Ensure that users of varying technological literacy levels can interact with the app effortlessly.
- **Real-time Order Management:** Implement a backend system that efficiently manages incoming orders, updating order statuses and sending notifications to customers when their orders are confirmed, being prepared, and ready for pickup/delivery.
- **Mobile Responsiveness:** Ensure that the web app is responsive and accessible across various devices, enabling customers to place orders conveniently from smartphones, tablets, and desktops.
- **Data Security and Privacy:** Implement robust security measures to safeguard customer data, ensuring compliance with data protection regulations.

By accomplishing these objectives, "GoFood – A Food Ordering App" aims to redefine the dining experience, eliminate the inconvenience of queuing, and bridge the gap between traditional dining practices and the convenience offered by cutting-edge technology.

CHAPTER-III

System Requirement Specification

This chapter presents the software and hardware requirements for the whole project. System requirements are essential in building a project. The system requirements help in identifying the resources required to execute the application. These requirements include hardware, software and other dependencies. By identifying the necessary resources, developers can ensure that the application runs smoothly, without any performance issues. System requirements also help in ensuring that the application is compatible with the intended hardware and software environment.

3.1 Visual Studio Code

Visual Studio Code (VS Code) emerges as a robust and highly beneficial tool for constructing web applications with the MERN stack. This section delves into the ways in which VS Code facilitates the development process in this context.

- **Integrated Development Environment (IDE):** VS Code stands as a comprehensive IDE that offers an array of features like IntelliSense, code debugging, and Git integration, all of which are highly advantageous when dealing with the complexities of a web project powered by the MERN stack.
- **Live Collaboration:** VS Code's Live Share extension empowers seamless real-time collaboration among developers. This functionality enables multiple team members to collaborate on the same codebase simultaneously, regardless of geographical constraints.
- **JavaScript and React Support:** VS Code provides extensive support for JavaScript and React development, including code-highlighting, autocompletion, and debugging. This capability proves invaluable when crafting the user interface and front-end components of the MERN stack application.
- **Multi-Platform Compatibility:** Being a cross-platform code editor, VS Code ensures that the development and testing of the MERN stack application can be seamlessly carried out on various operating systems, including Windows, macOS, and Linux.

- **Access to Libraries and Modules:** VS Code offers access to a wealth of JavaScript libraries and modules, streamlining the development process. This feature reduces the need to write every piece of code from scratch, enhancing productivity and code quality.
- **Integration with Other Tools:** VS Code seamlessly integrates with a variety of development tools, such as version control systems, project management platforms, and build tools. This integration fosters efficient project management and collaboration among team members, promoting a cohesive and coordinated workflow.

By harnessing the capabilities of Visual Studio Code within the MERN stack context, developers can create dynamic and scalable web applications while benefiting from a feature-rich environment that enhances productivity, collaboration, and code quality.

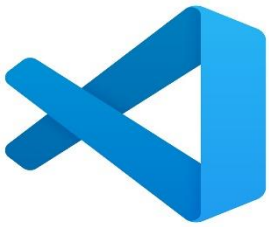


Figure 3.1: Visual Studio Code

3.2 Web Suite

- **HTML:** HTML is a markup language used to structure the content of web pages. It uses a system of tags to define the various elements within a web page, such as headings, paragraphs, lists, images, links, and more. These tags provide a logical framework for organizing information, making it possible for browsers to render content correctly. HTML forms the skeleton of a webpage, determining how content is laid out and structured.

Figure 3.2.1: HTML

- **CSS:** CSS is a style sheet language used to control the visual presentation of HTML elements. It defines how elements should be displayed, specifying attributes like colors, fonts, spacing, and layout. By using CSS, web designers and developers can create consistent and aesthetically pleasing designs for their websites. CSS allows for the separation of content and presentation, making it easier to maintain and update the appearance of a website across multiple pages.

Figure 3.2.2: CSS

3.3 React.js

React is an open-source JavaScript library widely used for building user interfaces (UIs) in web applications. Developed and maintained by Facebook, React allows developers to create dynamic and responsive UI components that efficiently update and render based on changes in data. React employs a declarative approach, where developers describe how a UI should look based on the application's state, and it automatically handles the efficient rendering of components as the data changes. One of React's key features is its virtual DOM (Document Object Model), which optimizes the process of updating the actual DOM, resulting in improved performance. React can be used in combination with other libraries and frameworks to create robust and interactive web applications.

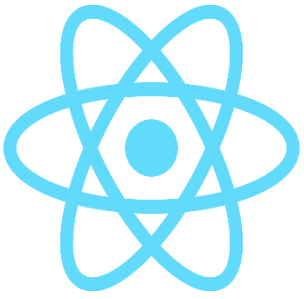


Figure 3.3.1: React.js

- **React Bootstrap:** React Bootstrap is a front-end framework that seamlessly blends the power of React, a popular JavaScript library for building user interfaces, with the versatility of Bootstrap, a widely used CSS framework. It offers a set of pre-designed UI components and responsive layout tools that enable developers to create modern, visually appealing, and user-friendly web applications with ease.

  At its core, React Bootstrap provides a collection of reusable components that are designed to fit seamlessly into React applications. These components cover a wide range of UI elements, including navigation bars, buttons, forms, modals, cards, alerts, and more. By incorporating these components, developers can expedite the development process, as they don't have to build each UI element from scratch.

  One of the major advantages of using React Bootstrap is its responsiveness. The framework is built with mobile-first design principles, ensuring that applications created with it are fully responsive and work smoothly on various screen sizes and devices. This responsiveness is achieved through the integration of Bootstrap's responsive grid system, which adapts the layout and components to different viewport sizes.

Figure 3.3.2: React Bootstrap

3.4 MongoDB and MongoDB Atlas

MongoDB stands as a prevalent open-source NoSQL database management system that adopts a flexible, scalable, and document-oriented approach to data storage. This approach is particularly advantageous for projects rooted in the MERN (MongoDB, Express.js, React, Node.js) stack, where dynamic data handling and evolving schemas are imperative. MongoDB thrives on collections of JSON-like documents, a design that enables agile data modeling and accommodates unstructured or semi-structured data.

Complementing MongoDB is MongoDB Atlas—an all-encompassing cloud-based database service provided by MongoDB Inc. This service revolutionizes the process of deploying, scaling, and managing MongoDB databases by automating manual tasks and simplifying complex configurations. MongoDB Atlas extends an array of features that enhance the overall database deployment experience.

In the context of the project to enhance the dining experience through the "GoFood – A Food Ordering App," MongoDB Atlas emerges as a central pillar in overseeing the numerous collections that house pivotal user data. From user particulars and login credentials to account insights, order history, and dining preferences, MongoDB Atlas stores a diverse spectrum of information. The cloud-based infrastructure of MongoDB Atlas empowers the application to seamlessly adapt to increased data volumes and surging user activity. This cloud-centric architecture, fortified by MongoDB Atlas's robust security mechanisms, assures the safeguarding of sensitive dining and financial data.

The high availability and automated backup mechanisms integral to MongoDB Atlas further reinforce data integrity, ensuring that the application remains resilient in the face of unforeseen events. This amalgamation of MongoDB and MongoDB Atlas not only furnishes a reliable and scalable data management framework but also aligns seamlessly with the requirements of the MERN stack's dynamic and user-centric development approach.

Figure 3.4: Mongo DB

3.5 Node.js and its frameworks

Node.js, is an open-source, server-side JavaScript runtime environment that allows developers to build scalable and high-performance network applications. It's based on the V8 JavaScript engine and uses an event-driven, non-blocking I/O model, making it well-suited for building real-time applications like chat applications, streaming services, and APIs. Node.js enables developers to use JavaScript on both the client and server sides, which streamlines the development process and promotes code reuse. It has a vast ecosystem of libraries and modules

available through the Node Package Manager (npm), making it a popular choice for building various types of applications.

Figure 3.5.1: Node.js

- **Express.js:** Express is a fast and minimalist web application framework for Node.js. It provides a set of tools and features that simplify the process of building web applications and APIs by offering a lightweight and flexible structure. Express enables developers to easily handle routing, middleware integration, and HTTP request/response handling. It's commonly used to create server-side applications, RESTful APIs, and other backend services in Node.js, allowing developers to focus on their application's logic rather than dealing with low-level HTTP intricacies.

Figure 3.5.2: Express.js

- **Mongoose:** Mongoose is an Object Data Modelling (ODM) library for MongoDB, a popular NoSQL database. It simplifies the interaction between Node.js applications and MongoDB by providing a structured and schema-based approach to data management. With Mongoose, developers can define models for their data, including schema definitions, data validation rules, and methods for interacting with the database. This abstraction layer streamlines tasks like CRUD operations, data manipulation, and querying, allowing developers to work with MongoDB in a more organized and consistent manner. Mongoose also provides features like middleware for handling pre and post operations on data, making it a powerful tool for building scalable and maintainable applications with MongoDB as the backend data store.

Figure 3.5.3: mongoose

- **Bcrypt:** bcrypt is a cryptographic library widely used for securely hashing passwords. It employs a slow and salted hashing algorithm, making it resistant to common password attacks. By incorporating a unique salt for each password, bcrypt significantly enhances security, thwarting rainbow table attacks and brute-force cracking attempts. It's a preferred choice for developers seeking to safeguard user passwords in databases, effectively mitigating the risks associated with data breaches and unauthorized access. Its deliberate computational intensity enhances password protection.

Figure 3.5.4: Bcrypt.js

- **JWT:** A JWT (JSON Web Token) is a compact way to securely send information between parties. It includes a header, a payload, and a signature. JWTs are often used for secure authentication and authorization in web applications. The header specifies how the token is encrypted, the payload holds the data, and the signature ensures the token's integrity. They are useful for sharing data without constant database queries.

Figure 2.5.5: JWT

- **Express Validator:** Express Validator is a middleware library for the Express web framework in Node.js. It simplifies the process of validating and sanitizing data received in HTTP requests. With built-in methods, it enables developers to define validation rules for request parameters, query strings, request bodies, and headers. This helps prevent malicious or incorrect data from entering the application, enhancing security and ensuring data integrity.

Figure 3.5.6: express-validator

- **Nodemon:** Nodemon is a development tool that monitors changes in files within a Node.js application and automatically restarts the server when changes are detected. This eliminates the need for manual server restarts during development, saving time and streamlining the testing process. Nodemon is particularly valuable for rapidly iterating and debugging applications, as it provides a seamless way to observe code changes and immediately see the effects without manual intervention. It's commonly used during the development phase to enhance productivity and improve the development experience.

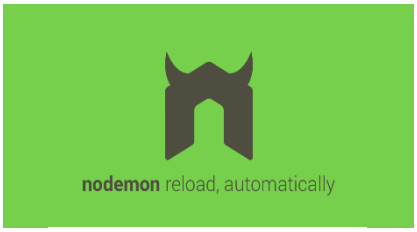


Figure 3.5.7: Nodemon

- **ThunderClient:** ThunderClient is a Visual Studio Code extension designed for efficient API testing. It empowers developers to create, manage, and execute HTTP requests within the VS Code environment. With a user-friendly interface, ThunderClient simplifies the process of configuring headers, parameters, and request bodies, while also facilitating response inspection and request debugging. It's a valuable tool for ensuring the functionality and reliability of APIs during development.

Figure 3.5.8: ThunderClient

CHAPTER-IV

Entity Relationship Diagram

The Entity-Relationship Diagram (ER Diagram) is a visual representation of thedata model used in the Personal Finance Tracker project. It illustrates the relationshipsbetween various entities (data objects) and their attributes. The ER Diagram serves as a vital tool for understanding the data structure and relationships within the application,helping developers and stakeholders grasp the project's data flow and organization. The ER Diagram of the project is as shown in Figure 4.1.

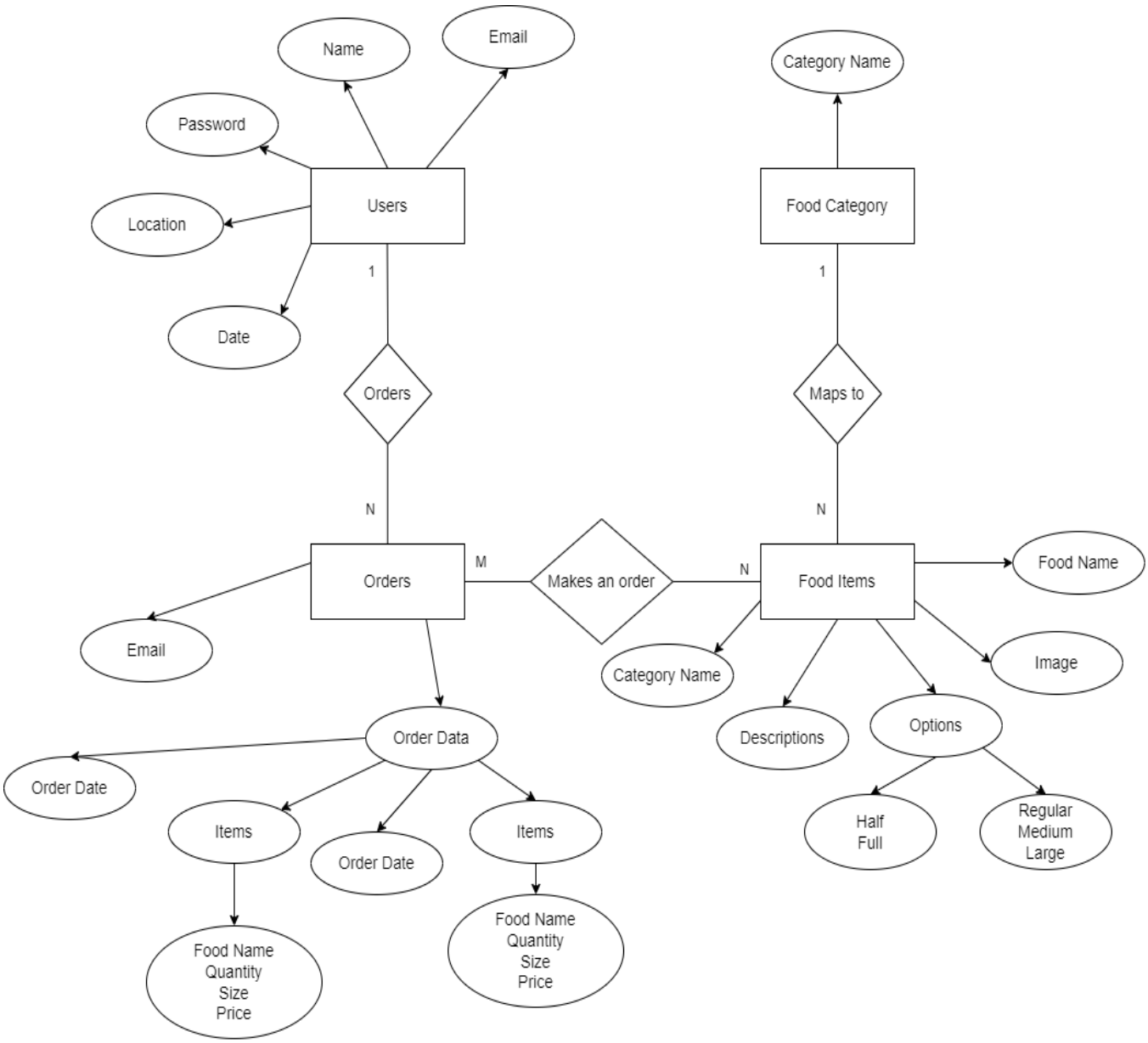


Figure 4.1: ER Diagram

CHAPTER-V

Schema Diagram

The Schema Diagram is a visual representation of the database structure used in the Personal Finance Tracker project. It provides a comprehensive view of the collections, documents, fields, and their relationships within the MongoDB database. The Schema Diagram is a crucial tool for understanding the organization and hierarchy of data in the application, assisting developers and stakeholders in grasping the data architecture. The Schema Diagram of the project is as shown in Figure 5.1.

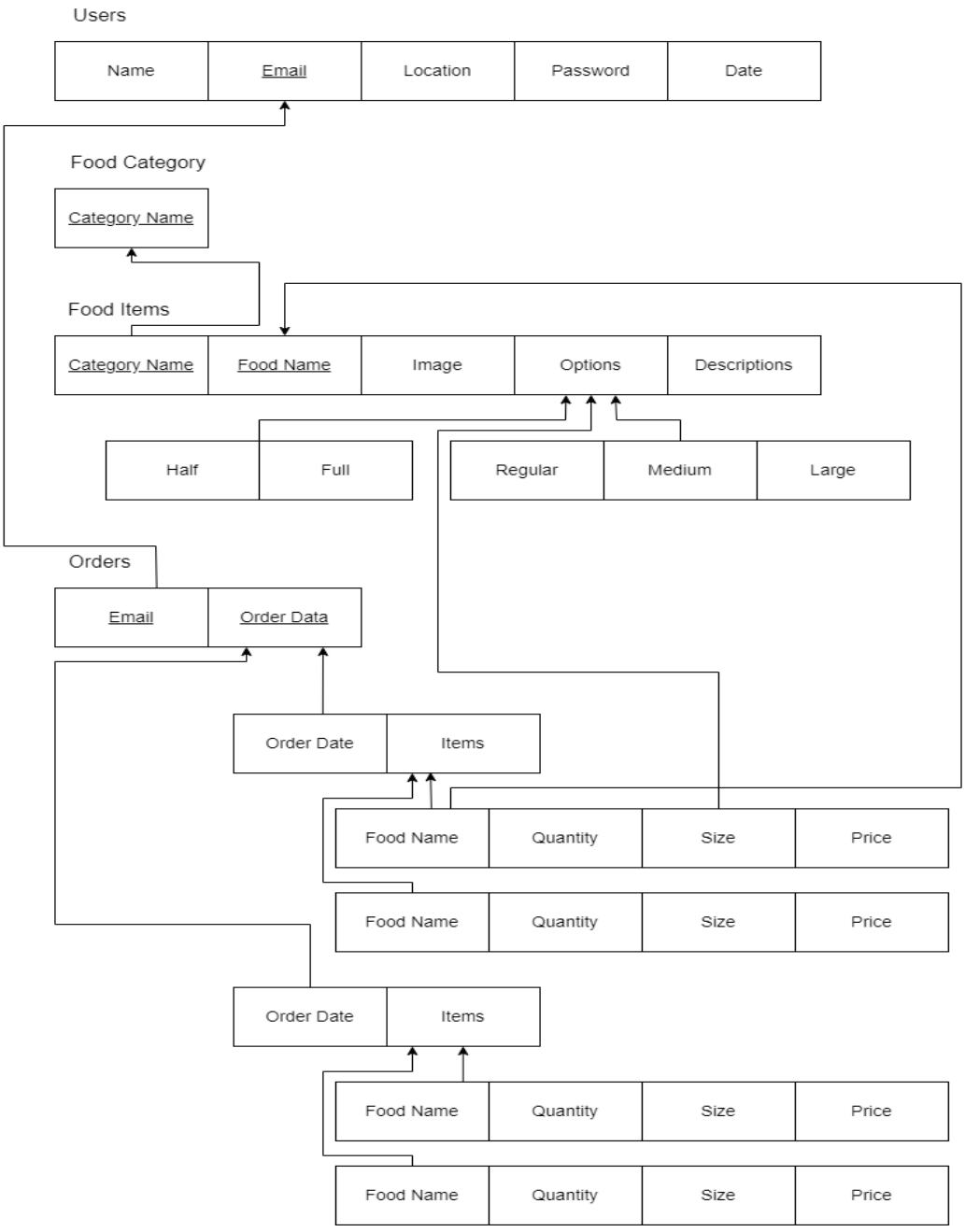


Figure 5.1: Schema Diagram

1. **User Collection:**

- **Name:** The "Name" attribute captures the individual's full name. It plays a pivotal role in personalizing user interactions and communications within the application. By including the first name and last name, the application can address users more intimately and create a user-centric environment.

- **Email:** The "Email" attribute acts as a unique identifier for each user within the system. This attribute ensures secure access and serves as a primary means of communication with users. It facilitates user authentication, enabling users to log in using their registered email addresses.

- **Location:** The "Location" attribute provides insights into the user's geographic location. While enhancing user profiles with location data. The location is captured using the geolocation function

- **Password:** The "Password" attribute represents a crucial aspect of user account security. Encrypted passwords are stored in this attribute, safeguarding user accounts from unauthorized access. This ensures that user data remains protected and only accessible by authenticated users.

- **Date:** The "Date" attribute signifies a timestamp associated with user interactions or registration. It aids in tracking user engagement and activity over time. For instance, the registration date could be used to recognize the user’s first registration to the web app.

2. **Food Category:**

- **Category Name:** The "Category Name" attribute captures the name or label of the food category. It provides a user-friendly representation of different culinary offerings, allowing users to easily identify and select their preferred categories from the menu.

3. **Food Items:**

- **Category Name:** The "Category Name" attribute establishes a link between food items and their respective categories. This attribute ensures that each food item is appropriately categorized, making it easier for users to navigate through the menu and find items of interest.

- **Food Name:** The "Food Name" attribute encapsulates the distinct name of each food item. This attribute provides a clear and recognizable label for the culinary offerings, enabling users to quickly identify and select their desired dishes.

- **Image:** The "Image" attribute contains a reference to an image that visually represents the food item. These images enhance the menu's visual appeal, offering users a tantalizing preview of the dishes. Visual cues contribute to an engaging and immersive dining experience.

- **Options:** The "Options" attribute presents a range of customization choices for each food item. This could include options such as "half," "full," "regular," "medium," and "large" portions. This attribute enables users to tailor their orders based on their appetite or preferences.

- **Description**: The "Description" attribute provides a brief overview of the food item. This description offers insights into the ingredients, preparation method, and flavor profile of the dish. Users can make informed decisions about their orders by referring to these descriptions.

4. **Orders:**

- **Email:** The "Email" attribute serves as a unique identifier for the customer who placed the order. This information ensures that each order is accurately linked to the corresponding user, facilitating personalized order history tracking and customer engagement.

- **Order Data:** The "Order Data" attribute encompasses a collection of arrays, each representing an individual order instance. Each array within the "Order Data" attribute encapsulates details related to the order date and the items ordered.

  - **Order Date:** The "Order Date" attribute records the specific date when the order was placed. This timestamp aids in organizing and tracking orders chronologically, providing insights into order patterns and customer preferences over time.
  - **Items Ordered:** Each order instance includes a list of food items that the customer selected. For each food item, attributes such as "id," "name," "quantity," "size," and "price" are captured.
  - **ID:** The "ID" attribute uniquely identifies each food item within the application's database. This ensures accuracy in associating the selected dishes with their corresponding entries in the menu.
  - **Name:** The "Name" attribute denotes the name of the food item that was ordered. This information provides a clear indication of the dishes included in the order.
  - **Quantity:** The "Quantity" attribute reflects the number of servings of a particular food item that were included in the order. This quantity information aids in managing inventory and tracking customer preferences.
  - **Size:** The "Size" attribute specifies the portion size chosen by the customer, such as "half." This customization option allows customers to tailor their orders based on their appetite or preferences.
  - **Price:** The "Price" attribute denotes the cost associated with the selected food item. This pricing information is crucial for calculating the total cost of the order and generating accurate billing statements.

CHAPTER-VI

Relational Tables

While the "GoFood – A Food Ordering App" project primarily utilizes a NoSQL database (MongoDB) for data storage, there are instances where relational structures are employed to establish connections between different data components. Relational tables offer a way to manage relationships between entities, enhancing data organization and integrity.The collections made for the project are displayed in Figure 6.1. "GoFood – A Food Ordering App" project judiciously balances NoSQL document storage with relational structures. While MongoDB excels at handling unstructured data, relational tables are employed where relationships and structured querying are key.

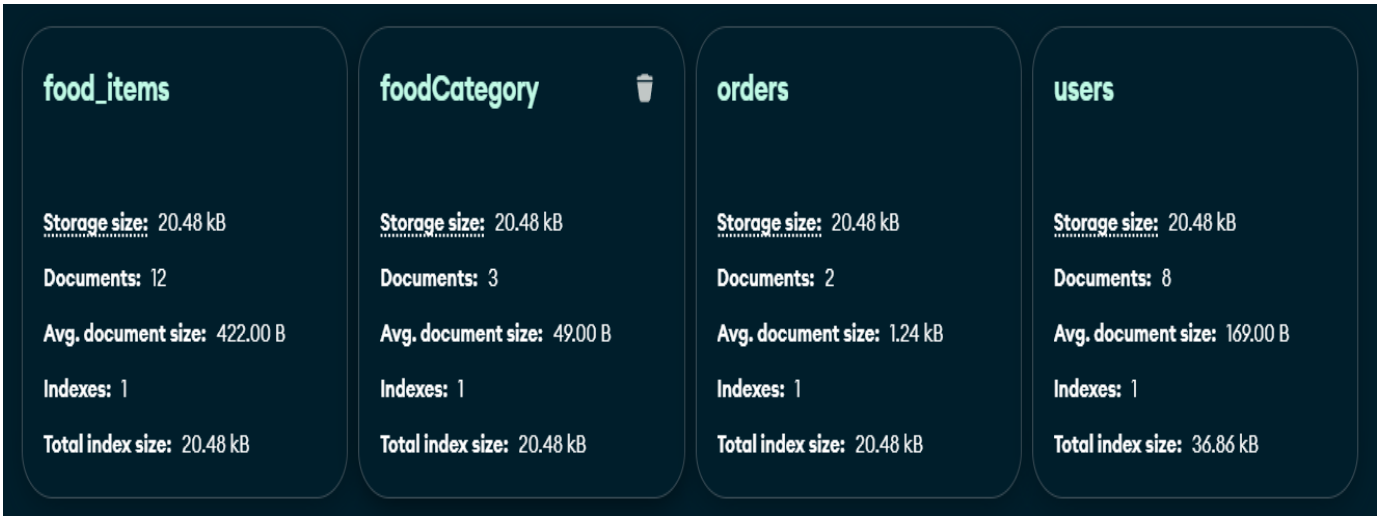
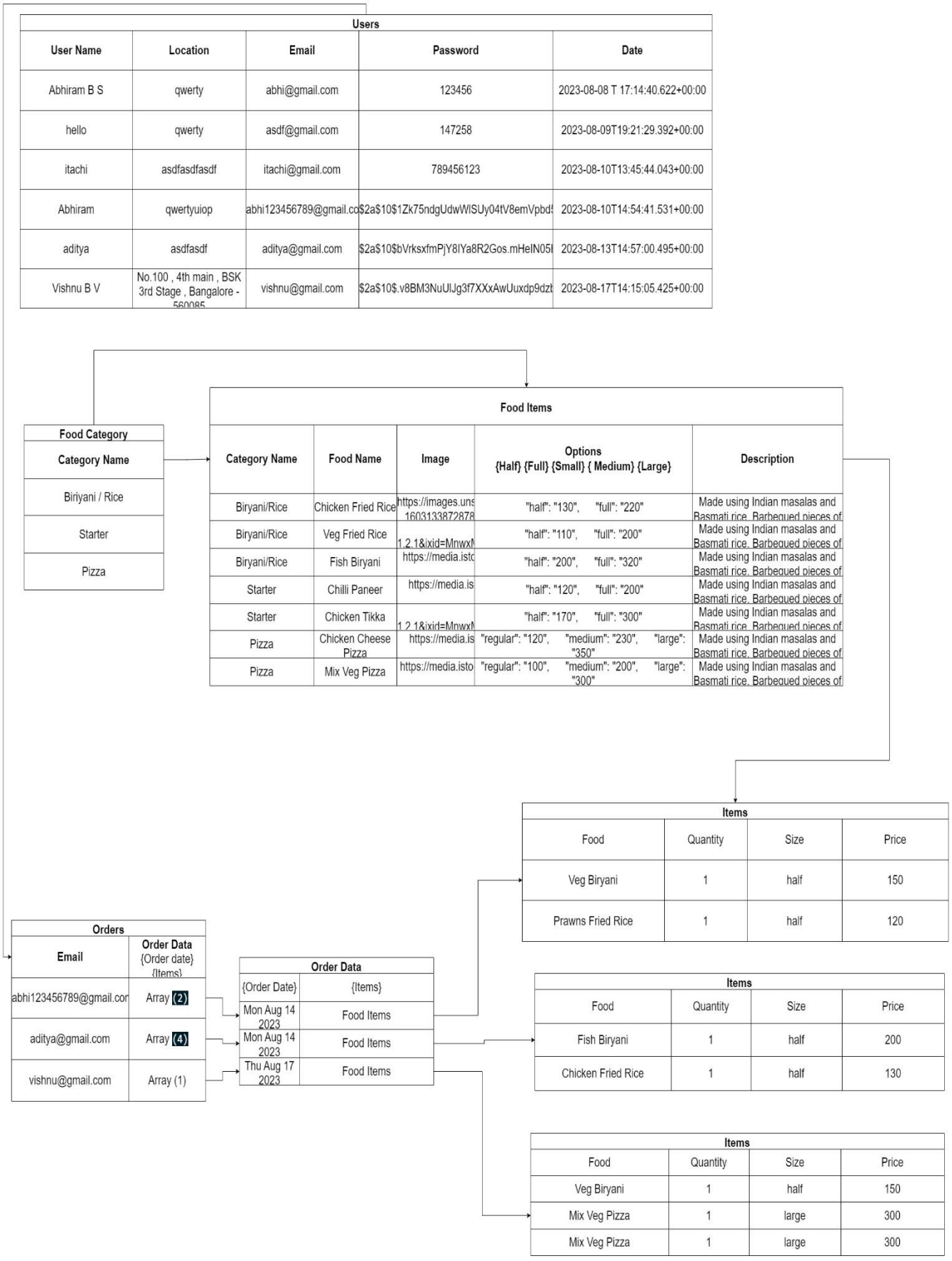


Figure 6.1.1: Collections in the Database

Even though the project predominantly employs MongoDB as a document-oriented database, certain project requirements necessitate a more structured approach for handling specific relationships. Relational tables are employed to effectively manage associations between users, their dining preferences and order history. While the project primarily centres around a document-oriented database, these relational structures contribute to enhanced data management and streamlined retrieval processes. An overview of the relational tables employed in the project is illustrated in Figure 6.2.



Users

User Name	Location	Email	Password	Date
Abhiram B S	qwerty	abhi@gmail.com	123456	2023-08-08 T 17:14:40.622+00:00
hello	qwerty	asdf@gmail.com	147258	2023-08-09T19:21:29.392+00:00
itachi	asdfasdfasdf	itachi@gmail.com	789456123	2023-08-10T13:45:44.043+00:00
Abhiram	qwertyuiop	abhi123456789@gmail.co	$2a$10$1Zk75ndgUdwWlSUy04tV8emVpbd	2023-08-10T14:54:41.531+00:00
aditya	asdfasdf	aditya@gmail.com	$2a$10$bVrksxfmPjY8lYa8R2Gos.mHeIN05	2023-08-13T14:57:00.495+00:00
Vishnu B V	No.100 , 4th main , BSK 3rd Stage , Bangalore - 560085	vishnu@gmail.com	$2a$10$.v8BM3NuUlJg3f7XXxAwUuxdp9dzt	2023-08-17T14:15:05.425+00:00

Food Category

Category Name
Biriyani / Rice
Starter
Pizza

Food Items

Category Name	Food Name	Image	Options {Half} {Full} {Small} { Medium} {Large}	Description
Biryani/Rice	Chicken Fried Rice	https://images.uns 1603133872878	"half": "130", "full": "220"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Biryani/Rice	Veg Fried Rice	1,2,1&ixid=MnwxM	"half": "110", "full": "200"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Biryani/Rice	Fish Biryani	https://media.istc	"half": "200", "full": "320"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Starter	Chilli Paneer	https://media.is	"half": "120", "full": "200"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Starter	Chicken Tikka	1,2,1&ixid=MnwxM	"half": "170", "full": "300"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Pizza	Chicken Cheese Pizza	https://media.is	"regular": "120", "medium": "230", "large": "350"	Made using Indian masalas and Basmati rice. Barbequed pieces of
Pizza	Mix Veg Pizza	https://media.isto	"regular": "100", "medium": "200", "large": "300"	Made using Indian masalas and Basmati rice. Barbequed pieces of

Orders

Email	Order Data {Order date} {Items}
abhi123456789@gmail.com	Array (2)
aditya@gmail.com	Array (4)
vishnu@gmail.com	Array (1)

Order Data

{Order Date}	{Items}
Mon Aug 14 2023	Food Items
Mon Aug 14 2023	Food Items
Thu Aug 17 2023	Food Items

Items

Food	Quantity	Size	Price
Veg Biryani	1	half	150
Prawns Fried Rice	1	half	120

Items

Food	Quantity	Size	Price
Fish Biryani	1	half	200
Chicken Fried Rice	1	half	130

Items

Food	Quantity	Size	Price
Veg Biryani	1	half	150
Mix Veg Pizza	1	large	300
Mix Veg Pizza	1	large	300

Figure 6.1.2: Relational Tables

CHAPTER-VII

Implementation Details

The implementation phase of the Food Ordering App project encompassed the development of critical components, including the backend logic, database integration, frontend interface, and the establishment of seamless communication between these elements. The project's execution was facilitated through the utilization of MongoDB as the data storage solution, Node.js for backend programming, Mongoose for database interactions, and React for constructing the frontend user interface.

7.1 MVC Architecture

The MVC (Model-View-Controller) architecture is a design pattern that serves as a guiding principle for structuring and organizing web applications. In the context of a MERN (MongoDB, Express.js, React, and Node.js) stack project, the MVC architecture provides a clear and efficient approach to managing the different components and functionalities of the application.

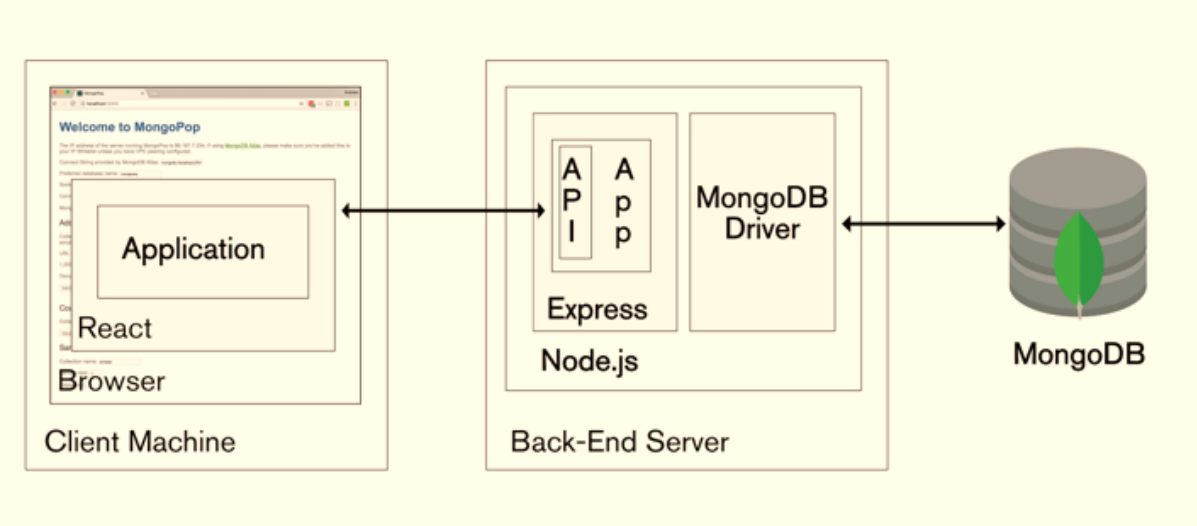


Figure 7.1: MVC Architecture

Model: The Model represents the underlying data structure and business logic of the application. In a MERN stack project, the Model encompasses the interaction with the MongoDB database. This involves defining data schemas and models that correspond to different entities within the application, such as users, orders, menu items, and more. The Model layer is responsible for tasks like data validation, CRUD (Create, Read, Update, Delete) operations, and ensuring data integrity.

View: The View layer is responsible for presenting data to users and capturing their interactions. In a MERN stack project, the View is primarily composed of the React components that form the user interface. These components render the data retrieved from the Model and provide an interactive and visually appealing experience for users. React's component-based architecture aligns seamlessly with the View layer's role, allowing for the creation of reusable and modular UI elements.

Controller: The Controller acts as an intermediary between the Model and the View. It handles user input and triggers appropriate actions within the application. In the MERN stack, the Controller is implemented using Express.js on the server-side. It defines routes, handles incoming requests, interacts with the Model to retrieve or manipulate data, and then communicates the relevant data to the View for rendering. The Controller ensures that the business logic is executed in response to user actions.

Interaction Flow: The MVC architecture promotes a clear separation of concerns, allowing different components to operate independently. In a MERN stack project, when a user interacts with the application, the flow typically follows this pattern:

- The user interacts with the View (React components) through the user interface.
- The View triggers a request that is sent to the Controller (Express.js) via an API route.
- The Controller processes the request, interacts with the Model (MongoDB), and retrieves or updates data as needed.
- The Controller sends the relevant data back to the View.
- The View (React components) renders the data and updates the user interface

.

Benefits:

The MVC architecture offers several benefits for MERN stack projects:

- **Modularity:** Components are separated into distinct layers, promoting modularity and ease of maintenance.
- **Scalability:** The clear separation of concerns facilitates scaling individual components as needed.
- **Code Reusability:** React components can be reused across different parts of the application, reducing redundant code.
- **Collaboration:** The division of responsibilities makes it easier for multiple developers to collaborate on different aspects of the project.

- **Testability:** Each component can be tested individually, enhancing the overall testing process.

In summary, the MVC architecture serves as a solid foundation for structuring MERN stack projects by ensuring a well-organized and efficient separation of concerns between data management (Model), user interface (View), and user interactions (Controller). This results in maintainable, scalable, and highly functional web applications.

7.2 Install all Node dependencies

Install all dependencies using the npm command for the frontend, it employs libraries like React, React Bootstrap, and React Router for user interface creation and navigation. On the backend, Express and Mongoose manage server-side operations, while packages like Axios and JWT enhance data handling and security. Additionally, testing tools like Testing Library aid in quality assurance. These dependencies collectively empower efficient development, ensuring seamless communication between frontend and backend components.

```
"dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "bootstrap": "^5.3.1",
    "bootstrap-dark-5": "^1.1.3",
    "react": "^18.2.0",
    "react-bootstrap": "^2.8.0",
    "react-dom": "^18.2.0",
    "react-router-dom": "^6.14.2",
    "react-scripts": "^5.0.1",
    "web-vitals": "^2.1.4"
  },
"dependencies": {
    "@emotion/react": "^11.11.1",
    "@emotion/styled": "^11.11.0",
    "@mui/icons-material": "^5.14.3",
    "@mui/material": "^5.14.4",
    "axios": "^1.4.0",
    "bcryptjs": "^2.4.3",
    "express": "^4.18.2",
    "express-validator": "^7.0.1",
    "jsonwebtoken": "^9.0.1",
    "mongoose": "^7.4.2",
    "nodemon": "^3.0.1"
  }
```

7.3 Backend Connection

The Backend connection was done using express.js. Here it initializes an Express server on port 5000, connecting to a MongoDB database. It configures CORS settings to allow requests from "http://localhost:3000" and handles incoming JSON data. The server defines routes using separate modules for creating users, displaying data, and managing orders. Upon execution, the server listens on the specified port, facilitating communication between the frontend and backend, enabling user registration, data display, and order management.

```
const express = require("express");
const app = express();
const port = 5000;
const mongoDB = require("./db");
mongoDB();

app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin", "http://localhost:3000");
  res.header(
    "Access-Control-Allow-Headers",
    "Origin, X-Requested-With, Content-Type, Accept"
  );
  next();
});

app.use(express.json());
app.use("/api", require("./Routes/CreateUser"));
app.use("/api", require("./Routes/DisplayData"));
app.use("/api", require("./Routes/OrderData"));
app.get("/", (req, res) => {
  res.send("Hello World!");
});

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`);
});
```

7.4 Database Integration

The Database Integration establishes a connection to a MongoDB database using the given URI. Upon successful connection, it fetches data from the "food_items" and "foodCategory" collections and stores them in global variables for application-wide access. The module exports the MongoDB connection function, encapsulating database connectivity and data retrieval. This snippet enhances data availability by connecting the application to the MongoDB database and retrieving relevant food item and category information for seamless integration into the Food Ordering App project.

```
const mongoose = require("mongoose");
const mongoURI =
  "mongodb://gofood:Abhi08032003@ac-jzhoilm-shard-00-
00.jcny2ll.mongodb.net:27017,ac-jzhoilm-shard-00-
01.jcny2ll.mongodb.net:27017,ac-jzhoilm-shard-00-
02.jcny2ll.mongodb.net:27017/gofoodmern?ssl=true&replicaSet=atlas-df53ns-
shard-0&authSource=admin&retryWrites=true&w=majority";

const MongoDB = async () => {
  try {
    await mongoose.connect(mongoURI, { useNewUrlParser: true });
    console.log("Connected to MongoDB");

    const fetched_data = await
mongoose.connection.db.collection("food_items");
    const data = await fetched_data.find({}).toArray();
    const foodCategory = await mongoose.connection.db.collection(
      "foodCategory"
    );
    const catData = await foodCategory.find({}).toArray();
    //console.log(data);
    global.food_items = data;
    global.foodCategory = catData;
    //console.log(global.food_items)
  } catch (err) {
    console.error("Error connecting to MongoDB:", err);
  }
};

module.exports = MongoDB;
```

7.5 Schema for User and Orders database

The below code snippet defines a MongoDB schema named "UserSchema" for the user collection. The schema comprises attributes like "name," "location," "email," "password," and "date." Each attribute is assigned a specific data type, and some attributes are marked as required or unique. The schema also includes a "date" attribute with a default value of the current date. The module exports the compiled model for the "user" collection, facilitating structured data storage and retrieval within the project's backend. This snippet establishes the blueprint for user data storage in MongoDB, supporting essential user information for authentication and personalization.

```
const mongoose = require("mongoose");

const { Schema } = mongoose;

const UserSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  location: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
    unique: true,
  },
  password: {
    type: String,
    required: true,
  },
  date: {
    type: Date,
    default: Date.now,
  },
});

module.exports = mongoose.model("user", UserSchema);
```

The below code snippet defines a MongoDB schema named "OrderSchema" for the order collection. The schema includes attributes such as "email" and "order_data." The "email" attribute holds the user's email address and is marked as required and unique, ensuring each order is associated with a distinct user. The "order_data" attribute is an array that captures the details of the order, encompassing ordered items, quantities, sizes, and prices. The module exports the compiled model for the "order" collection, facilitating structured data storage and retrieval. This snippet establishes a systematic structure to store order information, linking it to user emails in the MongoDB database.

```
const mongoose = require("mongoose");
const { Schema } = mongoose;
const OrderSchema = new Schema({
  email: {
    type: String,
    required: true,
    unique: true,
  },
  order_data: {
    type: Array,
    required: true,
  },
});
module.exports = mongoose.model("order", OrderSchema);
```

7.6 Frontend Creation and Routes

The frontend of the Food Ordering App includes a fundamental "Navbar" component developed using React. It features a responsive navigation bar with a collapsible menu that optimizes navigation. The recognizable app logo, "GoFood," also functions as a direct link to the homepage for user convenience. The collapsible menu dynamically presents options such as "Home" and "My Orders" (visible to authenticated users), facilitating effortless exploration of app sections. The component manages user authentication meticulously, offering "Login" and "Signup" buttons for unauthenticated users and "My Cart" with item count, along with a "Logout" option for authenticated users. Clicking on "My Cart" triggers a modal displaying the "Cart" component, while selecting "Logout" erases the authentication token, enhancing security. Another React-driven aspect is the "Cards" component, displaying individual food items and customizable preferences. It displays item images and names while allowing users to customize quantities and sizes using dropdown menus. The component dynamically computes the final price considering quantity and size choices, providing accurate cost estimates. Adding chosen items to the cart is a seamless process via the "Add to Cart" button, enhancing user engagement. The "Cards" component enriches the user experience by offering customization and easy item inclusion in the cart.Lastly, the React-based "Footer" component is integral to the frontend, providing consistent visual and informative support. Designed with responsiveness in mind, it presents crucial information in a structured manner. The footer incorporates the copyright notice "© 2023 GoFood, Inc.," strengthening the app's identity. Its adaptable design ensures visibility across various screen sizes, contributing to a polished appearance and user-friendly experience. Through an unobtrusive yet accessible display of copyright information, the "Footer" component enhances the overall app interface.

The provided code snippet configures the main application structure using React Router for navigation. It imports various screens such as "Home," "Login," "Signup," and "MyOrder," which correspond to different routes. Additionally, it integrates Bootstrap's dark theme for styling. The "CartProvider" component encapsulates application state related to the shopping cart. The App component wrapped in the "CartProvider" and contained within a Router, maps different routes to corresponding screen components. This snippet orchestrates navigation and rendering of screens based on route URLs, ensuring a seamless and dynamic user experience within the Food Ordering App.

```
import "./App.css";
import Home from "./screens/Home";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Login from "./screens/Login";
import "../node_modules/bootstrap-dark-5/dist/css/bootstrap-dark.min.css";
import "../node_modules/bootstrap/dist/js/bootstrap.bundle";
import "../node_modules/bootstrap/dist/js/bootstrap.bundle.min.js";
import Signup from "./screens/Signup";
import { CartProvider } from "./components/ContextReducer";
import MyOrder from "./screens/MyOrder";
function App() {
  return (
    <CartProvider>
      <Router>
        <div>
          <Routes>
            <Route exact path="/" element={<Home />} />
            <Route exact path="/login" element={<Login />} />
            <Route exact path="/createuser" element={<Signup />} />
            <Route exact path="/myOrder" element={<MyOrder />} />
          </Routes>
        </div>
      </Router>
    </CartProvider>
  );
}

export default App;
```

The below code snippet defines the "Home" component of the Food Ordering App's frontend using React. It fetches food categories and items from the backend API, allowing users to search for specific items. The component features a responsive carousel displaying food images, a search bar, and dynamically rendered cards for each food item within their respective categories. Users can filter and view food items based on search queries and categories. Overall, this component offers an engaging user interface for browsing and selecting food items, enhancing the user experience in the Food Ordering App.

```
return (
    <div>
      <div>
        <Navbar></Navbar>
      </div>
      <div>
        <div
          id="carouselExampleFade"
          className="carousel slide carousel-fade"
          data-bs-ride="carousel"
          style={{ objectFit: "contain !important" }}
        >
          <div className="carousel-inner" id="carousel">
            <div className="carousel-caption" style={{ zIndex: "15" }}>
              <div className="d-flex justify-content-center">
                <input
                  className="form-control me-2"
                  type="search"
                  placeholder="Search"
                  aria-label="Search"
                  value={search}
                  onChange={(e) => {
                    setSearch(e.target.value);
                  }}
                />
              </div>
            </div>
            <div className="carousel-item active">
              <img
                src="https://source.unsplash.com/random/300×300/?burger"
                className="d-block w-100"
                style={{ filter: "brightness(50%)" }}
                alt="..."
              />
            </div>
            <div className="carousel-item">
              <img
                src="https://source.unsplash.com/random/300×300/?barbeque"
                className="d-block w-100"
                style={{ filter: "brightness(50%)" }}
                alt="..."
              />
            </div>
            <div className="carousel-item">
              <img
                src="https://source.unsplash.com/random/300×300/?pastry"
                className="d-block w-100"
                style={{ filter: "brightness(50%)" }}
                alt="..."
              />
            </div>
          </div>
          <button
            className="carousel-control-prev"
            type="button"
            data-bs-target="#carouselExampleFade"
            data-bs-slide="prev"
          >
            <span
              className="carousel-control-prev-icon"
              aria-hidden="true"
            ></span>
            <span className="visually-hidden">Previous</span>
          </button>
          <button
            className="carousel-control-next"
            type="button"
            data-bs-target="#carouselExampleFade"
            data-bs-slide="next"
          >
            <span
              className="carousel-control-next-icon"
              aria-hidden="true"
            ></span>
            <span className="visually-hidden">Next</span>
          </button>
        </div>
      </div>
```

```
      <div className="container">
        {foodCat !== []
          ? foodCat.map((data) => {
              return (
                <div className="row mb-3">
                  <div key={data._id} className="fs-3 m-3">
                    {data.CategoryName}
                  </div>
                  <hr />
                  {foodItem !== [] ? (
                    foodItem
                      .filter(
                        (item) =>
                          item.CategoryName === data.CategoryName &&
item.name.toLowerCase().includes(search.toLowerCase())
                      )
                      .map((filterItems) => {
                        return (
                          <div
                            key={filterItems._id}
                            className="col-12 col-md-6 col-lg-3"
                          >
                            <Card
                              foodItem={filterItems}
                              options={filterItems.options[0]}
                            ></Card>
                          </div>
                        );
                      })
                  ) : (
                    <div>No Such Data Found</div>
                  )}
                </div>
              );
            })
          : ""}
      </div>
      <div>
        <Footer></Footer>
      </div>
    </div>
  );
```

The provided code snippet defines the "Signup" and "login" component in the Food Ordering App's frontend using React. This component displays a user registration form with fields for name, email, password, and geolocation. Users can input their information and submit the form to create an account. The form also offers the option to fetch the user's current location. Upon submission, the data is sent to the backend API for user registration. The component features a visually appealing design with background images, enhancing the user interface and facilitating the registration process for new users in the Food Ordering App.

```jsx
import React, { useState } from "react";
import { Link } from "react-router-dom";
import Navbar from "../components/Navbar";

export default function Signup() {
  const [credentials, setcredentials] = useState({
    name: "",
    email: "",
    password: "",
    geolocation: "",
  });

  const handleSubmit = async (e) => {
    e.preventDefault();
    const response = await fetch("http://localhost:5000/api/createuser", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        name: credentials.name,
        email: credentials.email,
        password: credentials.password,
        location: credentials.geolocation,
      }),
    });
    const json = await response.json();
    console.log(json);
    if (!json.success) {
      alert("Enter Valid Credentials");
    }
  };
  const onChange = (event) => {
    setcredentials({ ...credentials, [event.target.name]: event.target.value
});
  };

  return (
    <div
      style={{
        backgroundImage:
          'url("https://images.pexels.com/photos/1565982/pexels-photo-
1565982.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1")',
        backgroundSize: "cover",
        height: "100vh",
      }}
    >
      <div>
        <Navbar />
      </div>

      <div className="container">
        <form
          className="w-50 m-auto mt-5 border bg-dark border-success rounded"
          onSubmit={handleSubmit}
        >
          <div className="m-3">
            <label htmlFor="name" className="form-label">
              Name
            </label>
            <input
              type="text"
              className="form-control"
              name="name"
              value={credentials.name}
              onChange={onChange}
            />
          </div>

          <div className="m-3">
            <label htmlFor="exampleInputEmail1" className="form-label">
              Email address
            </label>
            <input
              type="email"
              className="form-control"
              name="email"
              value={credentials.email}
              onChange={onChange}
              id="exampleInputEmail1"
              aria-describedby="emailHelp"
            />
            <div id="emailHelp" className="form-text">
              We'll never share your email with anyone else.
            </div>
          </div>

          <div className="m-3">
            <label htmlFor="exampleInputPassword1" className="form-label">
              Password
            </label>
            <input
              type="password"
              className="form-control"
              name="password"
              value={credentials.password}
              onChange={onChange}
              id="exampleInputPassword1"
            />
          </div>

          <div className="m-3">
            <label htmlFor="exampleInputPassword1" className="form-label">
              Address
            </label>
            <input
              type="text"
              className="form-control"
              name="geolocation"
              value={credentials.geolocation}
              onChange={onChange}
              id="exampleInputPassword1"
            />
            <div className="m-3">
              <button
                type="button"
                name="geolocation"
                className=" btn btn-success"
              >
                Click for current Location{" "}
              </button>
            </div>
          </div>

          <button type="submit" className="m-3 btn btn-success">
            Submit
          </button>
          <Link to="/login" className="m-3 btn btn-danger">
            Already a user
          </Link>
        </form>
      </div>
    </div>
  );
}
```

```jsx
import React, { useState } from "react";
import Navbar from "../components/Navbar";
import { Link, useNavigate } from "react-router-dom";

export default function Login() {
  const [credentials, setcredentials] = useState({
    email: "",
    password: "",
  });
  let navigate = useNavigate();

  const handleSubmit = async (e) => {
    e.preventDefault();
    const response = await fetch("http://localhost:5000/api/loginuser", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        email: credentials.email,
        password: credentials.password,
      }),
    });
    const json = await response.json();
    console.log(json);
    if (!json.success) {
      alert("Enter Valid Credentials");
    }
    if (json.success) {
      localStorage.setItem("userEmail", credentials.email);
      localStorage.setItem("authToken", json.authToken);
      console.log(localStorage.getItem("authToken"));
      navigate("/");
    }
  };
  const onChange = (event) => {
    setcredentials({ ...credentials, [event.target.name]: event.target.value
});
  };
  return (
    <div
      style={{
        backgroundImage:
          'url("https://images.pexels.com/photos/326278/pexels-photo-
326278.jpeg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=1")',
        height: "100vh",
        backgroundSize: "cover",
      }}
    >
      <div>
        <Navbar />
      </div>
      <div className="container">
        <form
          className="w-50 m-auto mt-5 border bg-dark border-success rounded"
          onSubmit={handleSubmit}
        >
          <div className="m-3">
            <label htmlFor="exampleInputEmail1" className="form-label">
              Email address
            </label>
            <input
              type="email"
              className="form-control"
              name="email"
              value={credentials.email}
              onChange={onChange}
              id="exampleInputEmail1"
              aria-describedby="emailHelp"
            />
            <div id="emailHelp" className="form-text">
              We'll never share your email with anyone else.
            </div>
          </div>

          <div className="m-3">
            <label htmlFor="exampleInputPassword1" className="form-label">
              Password
            </label>
            <input
              type="password"
              className="form-control"
              name="password"
              value={credentials.password}
              onChange={onChange}
              id="exampleInputPassword1"
            />
          </div>

          <button type="submit" className="m-3 btn btn-success">
            Submit
          </button>
          <Link to="/createuser" className="m-3 btn btn-danger">
            I'm a new user
          </Link>
        </form>
      </div>
    </div>
  );
}
```

```
import React from "react";
import { useCart, useDispatchCart } from "../components/ContextReducer";
import trash from "../trash.svg";

export default function Cart() {
  let data = useCart();
  let dispatch = useDispatchCart();
  if (data.length === 0) {
    return (
      <div>
        <div className="m-5 w-100 text-center fs-3">Your cart is empty</div>
      </div>
    );
  }
  const handleCheckOut = async () => {
    let userEmail = localStorage.getItem("userEmail");
    let response = await fetch("http://localhost:5000/api/OrderData", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        order_data: data,
        email: userEmail,
        order_date: new Date().toDateString(),
      }),
    });
    console.log("Order Response:", response);
    if (response.status === 200) {
      dispatch({ type: "DROP" });
    }
  };
  let totalPrice = data.reduce((total, food) => total + food.price, 0);
  return (
    <div>
      <div className="container m-auto mt-5 table-responsive  table-
responsive-sm table-responsive-md">
        <table className="table table-hover ">
          <thead className=" text-success fs-4">
            <tr>
              <th scope="col">#</th>
              <th scope="col">Name</th>
              <th scope="col">Quantity</th>
              <th scope="col">Option</th>
              <th scope="col">Amount</th>
              <th scope="col"></th>
            </tr>
          </thead>
          <tbody>
            {data.map((food, index) => (
              <tr>
                <th scope="row">{index + 1}</th>
                <td>{food.name}</td>
                <td>{food.qty}</td>
                <td>{food.size}</td>
                <td>{food.price}</td>
                <td>
                  <button type="button" className="btn p-0">
                    <img
                      src={trash}
                      alt="delete"
                      onClick={() => {
                        dispatch({ type: "REMOVE", index: index });
                      }}
                    ></img>
                  </button>
                </td>
              </tr>
            ))}
          </tbody>
        </table>
        {
          <div>
            <h1 className="fs-2">Total Price: {totalPrice}/-</h1>
          </div>
        }
        <div>
          <button className="btn bg-success mt-5 " onClick={handleCheckOut}>
            Check Out
          </button>
        </div>
      </div>
    </div>
  );
}
```

The above code snippet shows the "Cart" component in the Food Ordering App's frontend is defined using React. This component displays the user's selected food items in a table format, along with their respective quantities, options, prices, and a delete option. The component calculates the total price of the items in the cart and provides a "Check Out" button to complete the order. Users can remove items from the cart using the trash icon, which updates the cart in real-time. The design is responsive, ensuring optimal viewing across different devices. This component enhances the user experience by providing a clear overview of selected items, their details, and a seamless checkout process for the Food Ordering App.

The below code snippet shows the "MyOrder" component in the Food Ordering App's frontend is implemented using React. This component retrieves and displays a user's order history. It fetches the user's order data from the backend API using the user's email stored in local storage. The fetched data is then processed to display a chronological list of orders, including the order date, food items, quantities, sizes, and prices. The design ensures that orders are displayed clearly and grouped by their respective order dates, providing users with a convenient way to track their past orders. The component is responsive and integrates with the Navbar and Footer components for a cohesive user experience in the Food Ordering App.

This code snippet establishes a context for managing a shopping cart's state in a React application. It defines two contexts, CartStateContext and CartDispatchContext, along with a reducer function that handles different actions for modifying the cart state. The reducer supports actions like "ADD" for adding items, "REMOVE" for removing items, "UPDATE" for updating item quantities and prices, and "DROP" for clearing the cart. The CartProvider component wraps the application with the cart state and dispatch contexts, allowing components to access and manage the cart state using the useCart and useDispatchCart custom hooks. Overall, this code facilitates centralized management of the shopping cart's state and actions within the application.

```
import React, { useEffect, useState } from "react";
import Footer from "../components/Footer";
import Navbar from "../components/Navbar";

export default function MyOrder() {
  const [orderData, setOrderData] = useState("");

  const fetchMyOrder = async () => {
    console.log(localStorage.getItem("userEmail"));
    await fetch("http://localhost:5000/api/myOrderData", {
      // credentials: 'include',
      // Origin:"http://localhost:3000/login",
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        email: localStorage.getItem("userEmail"),
      }),
    }).then(async (res) => {
      let response = await res.json();
      await setOrderData(response);
    });

    // await res.map((data)=>{
    //     console.log(data)
    // })
  };

  useEffect(() => {
    fetchMyOrder();
  }, []);
  return (
    <div>
      <div>
        <Navbar></Navbar>
      </div>
      <div classNameName="container">
        <div classNameName="row">
          {orderData !== {}
            ? Array(orderData).map((data) => {
                return data.orderData
                  ? data.orderData.order_data
                      .slice(0)
                      .reverse()
                      .map((item) => {
                        return item.map((arrayData) => {
                          return (
                            <div>
                              {arrayData.Order_date ? (
                                <div classNameName="m-auto mt-5">
                                  {(data = arrayData.Order_date)}
                                  <hr />
                                </div>
                              ) : (
                                <div classNameName="col-12 col-md-6 col-lg-3">
                                  <div
                                    classNameName="card mt-3"
                                    style={{
                                      width: "16rem",
                                      maxHeight: "360px",
                                    }}
                                  >
                                    {/* <img
                                      src={arrayData.img}
                                      classNameName="card-img-top"
                                      alt="..."
                                      style={{
                                        height: "120px",
                                        objectFit: "fill",
                                      }}
                                    /> */}
                                    <div classNameName="card-body">
                                      <h5 classNameName="card-title">
                                        {arrayData.name}
                                      </h5>
                                      <div
                                        classNameName="container w-100 p-0"
                                        style={{ height: "38px" }}
                                      >
                                        <span classNameName="m-1">
                                          {arrayData.qty}
                                        </span>
                                        <span classNameName="m-1">
                                          {arrayData.size}
                                        </span>
                                        <span classNameName="m-1">{data}</span>
                                        <div classNameName=" d-inline ms-2 h-100 w-20 fs-5">
                                          ₹{arrayData.price}/-
                                        </div>
                                      </div>
                                    </div>
                                  </div>
                                </div>
                              )}
                            </div>
                          );
                        });
                      })
                  : "";
              })
            : ""}
        </div>
      </div>

      <div>
        <Footer></Footer>
      </div>
    </div>
  );
}
```

```jsx
import React, { createContext, useContext, useReducer } from "react";

const CartStateContext = createContext();
const CartDispatchContext = createContext();

const reducer = (state, action) => {
  switch (action.type) {
    case "ADD":
      return [
        ...state,
        {
          id: action.id,
          name: action.name,
          qty: action.qty,
          size: action.size,
          price: action.price,
          img: action.img,
        },
      ];
    case "REMOVE":
      let newArr = [...state];
      newArr.splice(action.index, 1);
      return newArr;
    case "UPDATE":
      let arr = [...state];
      arr.find((food, index) => {
        if (food.id === action.id) {
          console.log(
            food.qty,
            parseInt(action.qty),
            action.price + food.price
          );
          arr[index] = {
            ...food,
            qty: parseInt(action.qty) + food.qty,
            price: action.price + food.price,
          };
        }
        return arr;
      });
      return arr;
    case "DROP":
      let empArray = [];
      return empArray;

    default:
      console.log("Error in Reducer");
  }
};

export const CartProvider = ({ children }) => {
  const [state, dispatch] = useReducer(reducer, []);
  return (
    <CartDispatchContext.Provider value={dispatch}>
      <CartStateContext.Provider value={state}>
        {children}
      </CartStateContext.Provider>
    </CartDispatchContext.Provider>
  );
};
export const useCart = () => useContext(CartStateContext);
export const useDispatchCart = () => useContext(CartDispatchContext);
```

7.7 User Authentication and Management

This code snippet represents an Express.js router that handles user registration and login functionalities for an application. The /createuser route is used for registering users, validating the provided email, name, and password. The password is securely hashed before storing in the database. The /loginuser route handles user login, checking the email and password against the database. If credentials are valid, a JSON Web Token (JWT) is generated, allowing authorized access to the application. Errors and success responses are appropriately managed, and the router is exported for integration into the application's backend. This code snippet provides a secure and structured approach to user registration and login with password hashing and JWT authentication.

The Express.js router responsible for handling order-related operations in a backend system. The /orderData route is designed to process incoming order data. It either creates a new order entry associated with a user's email or appends new order information to an existing user's orders. The data includes order items and the order date. The /myorderData route is used to retrieve a user's order history based on their email. These routes interact with a database model named "Orders" to manage user orders effectively. Appropriate error handling and responses are incorporated to ensure smooth functionality. The code snippet forms a robust backend solution for order creation and retrieval in an application.

The Express.js router handling a POST request to retrieve food-related data. The route, named /foodData, is responsible for serving food item information and categories from a global context. The code sends a response containing an array containing the food items and their corresponding categories. This route acts as a backend endpoint to supply food-related data to the frontend of an application. Any potential errors are caught and logged, with a "Server Error" response sent if an issue arises. The code plays a crucial role in providing necessary data for the frontend's food-related functionalities.

```
const express = require("express");
const router = express.Router();
const User = require("../models/User");
const { body, validationResult } = require("express-validator");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const jwtSecret = "MynameisAbhiramBS";

router.post(
  "/createuser",
  [
    body("email").isEmail(),
    body("name").isLength({ min: 5 }),
    body("password", "Incorrect Password").isLength({ min: 5 }),
  ],
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    const salt = await bcrypt.genSalt(10);
    let secPassword = await bcrypt.hash(req.body.password, salt);

    try {
      await User.create({
        name: req.body.name,
        password: secPassword,
        email: req.body.email,
        location: req.body.location,
      });
      res.json({ success: true });
    } catch (error) {
      console.log(error);
      res.json({ success: false });
    }
  }
);

router.post(
  "/loginuser",
  [
    body("email").isEmail(),
    body("password", "Incorrect Password").isLength({ min: 5 }),
  ],
  async (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
    }

    let email = req.body.email;
    try {
      let userData = await User.findOne({ email });
      if (!userData) {
        return res
          .status(400)
          .json({ errors: "Try logging in with correct credentials" });
      }
      const pwdCompare = await bcrypt.compare(
        req.body.password,
        userData.password
      );

      if (!pwdCompare) {
        return res
          .status(400)
          .json({ errors: "Try logging in with correct credentials" });
      }
      const data = {
        user: {
          id: userData.id,
        },
      };
      const authToken = jwt.sign(data, jwtSecret);
      return res.json({ success: true, authToken: authToken });
    } catch (error) {
      console.log(error);
      res.json({ success: false });
    }
  }
);

module.exports = router;
```

7.8 Testing and Debugging

Testing and debugging using ThunderClient, a versatile API client extension for Visual Studio Code, involves a streamlined and efficient process to ensure the functionality and performance of APIs. ThunderClient provides a user-friendly interface to create, send, and analyze HTTP requests, making it an excellent tool for developers.

In the testing phase, ThunderClient simplifies the creation of requests by offering intuitive input fields for various parameters like headers, query parameters, and request bodies. This facilitates comprehensive testing of API endpoints with different inputs. It supports various HTTP methods, aiding in testing different scenarios such as GET, POST, PUT, DELETE, etc. Test suites can be organized, enabling the execution of multiple requests in a sequence, emulating real-world scenarios.

The debugging process is made more efficient by ThunderClient's real-time response visualization. Responses are presented in a well-structured format, making it easier to spot errors or discrepancies. Debugging is further enhanced through ThunderClient's ability to display headers, response time, status codes, and response body in a clear manner.

ThunderClient's interactive interface allows developers to modify requests on the fly and observe how changes impact responses. This feature expedites debugging by enabling immediate adjustments and iterations to ensure proper functionality. Additionally, ThunderClient supports exporting and importing requests, facilitating collaboration and documentation.

In summary, ThunderClient simplifies testing and debugging by providing an intuitive interface, real-time response visualization, and on-the-fly request modifications. Its capabilities enhance the efficiency and accuracy of identifying and rectifying issues within APIs, contributing to the overall quality and reliability of web applications.

7.9 Responsiveness and User Experience

The "GoFood – A Food Ordering App" project is designed to elevate user experience by integrating the MongoDB, Express.js, React, and Node.js technologies. It emphasizes creating a seamless and engaging platform for users. Leveraging React's dynamic components and state management, the interface offers intuitive navigation and real-time updates. Express.js enables efficient data handling and API integration, enhancing responsiveness. MongoDB stores and retrieves data, ensuring scalability and robustness. The project optimizes user interactions through responsive design, adapting content to diverse screen sizes. User-centric features, like forms and personalized recommendations, enrich engagement. Comprehensive testing and debugging ensure a smooth user journey. Overall, the MERN project prioritizes user satisfaction by seamlessly combining these technologies, resulting in a sophisticated, dynamic, and user-friendly web application that delivers a highly satisfying and immersive user experience.

CHAPTER-VIII

Results

8.1 Webpage Launch

Upon accessing the webpage, users are immediately directed to the homepage of the "GoFood – A Food Ordering App." Here, they encounter the fundamental layout of the app, featuring key components such as the Navigation Bar, Carousel, Cards, and Footer. Notably, the Navbar prominently offers the options for both Sign Up and Login, facilitating user interaction and access to the app's functionalities. The homepage is shown in the below images.

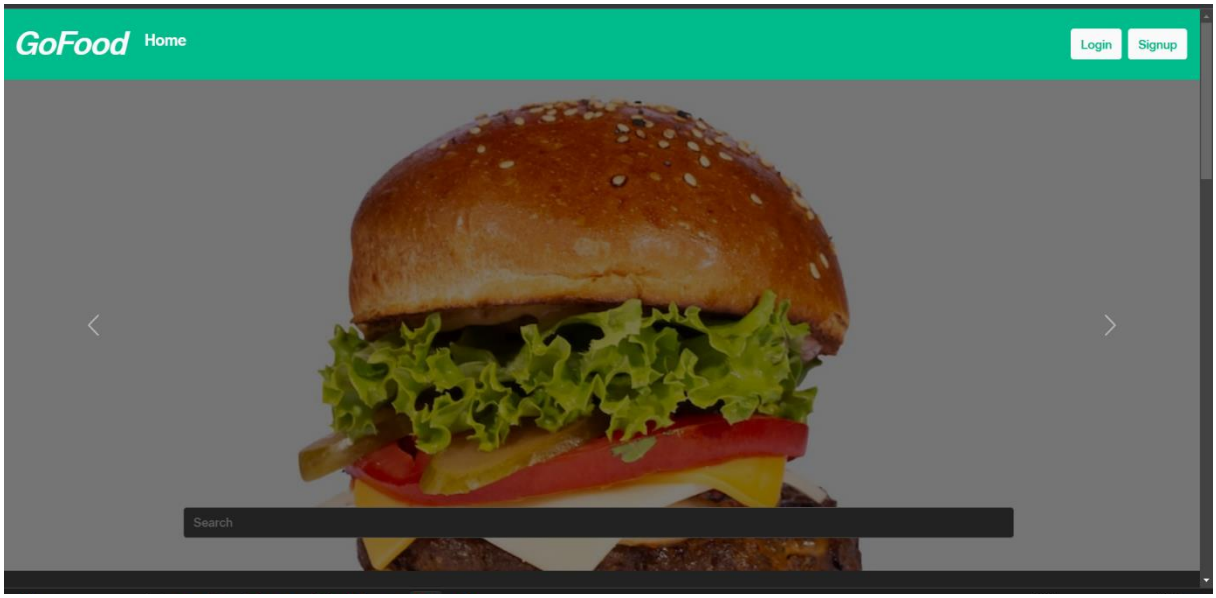


Figure 8.1.1: Display of Home Page

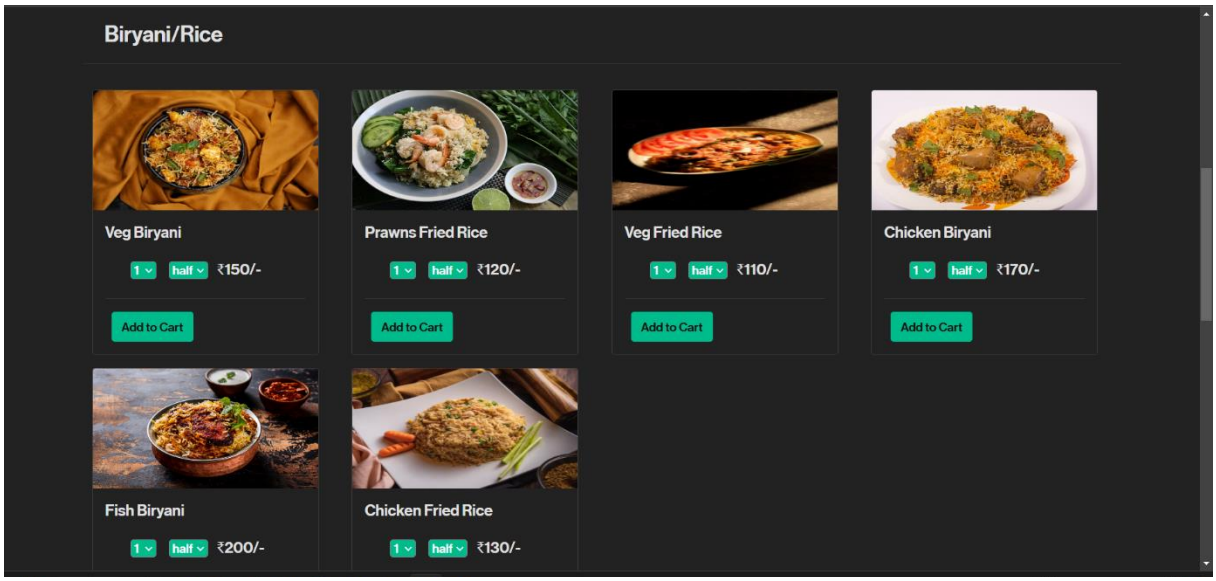


Figure 8.1.2: Display of Food Items According to Food Category

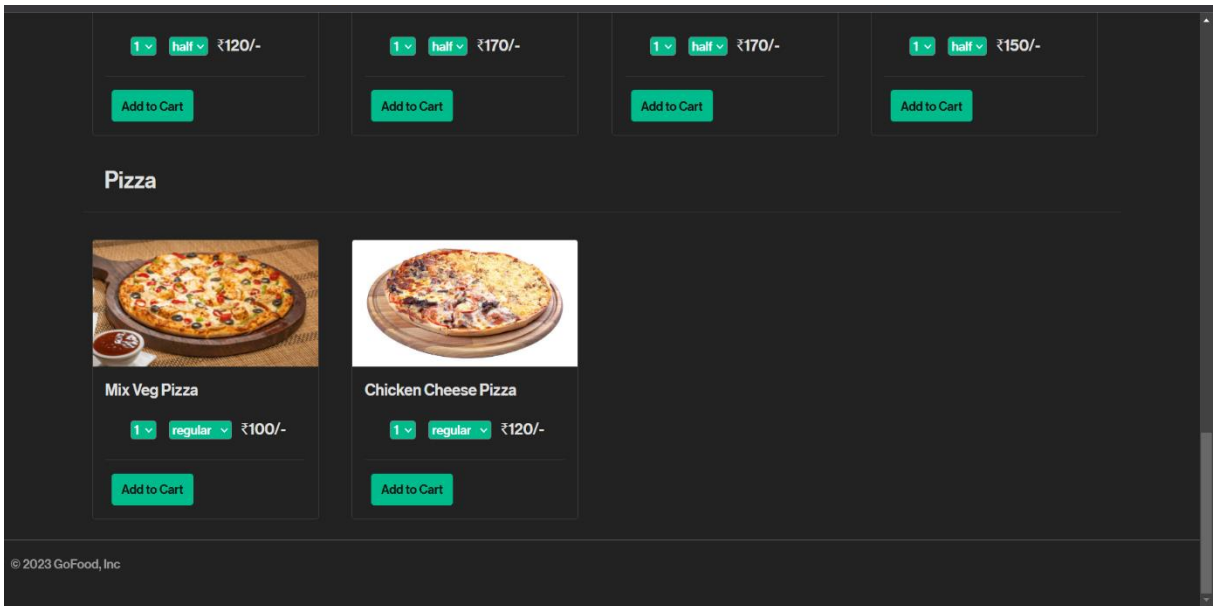


Figure 8.1.3: Display of Footer

8.2 SignUp and Login Page

For new users, the initial step involves navigating to the SignUp page to create an account. This process entails entering essential details like Username, Email address, Password, and Location for successful registration. Once registered, users can subsequently access the web app by logging in. The Login process entails entering the registered Email and Password. All the Data entered by the user are stored in the database. Visual representations of the Signup and Login Pages are provided below for reference.

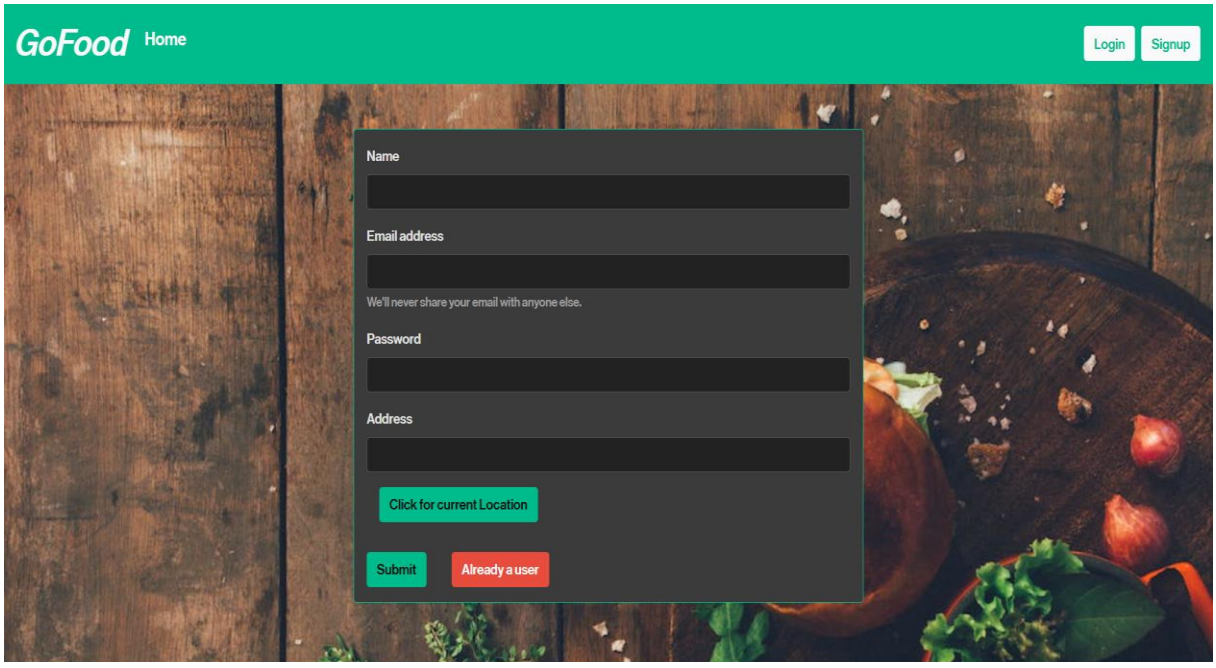


Figure 8.2.1: Signup Page for New users

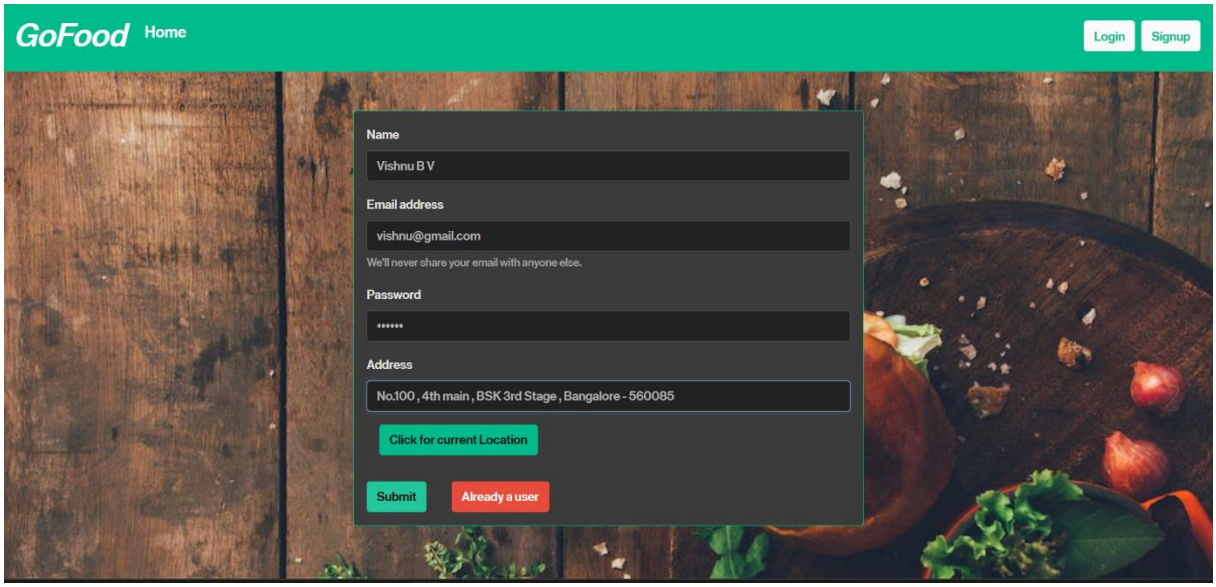


Figure 8.2.2: SignUp Process for a new user

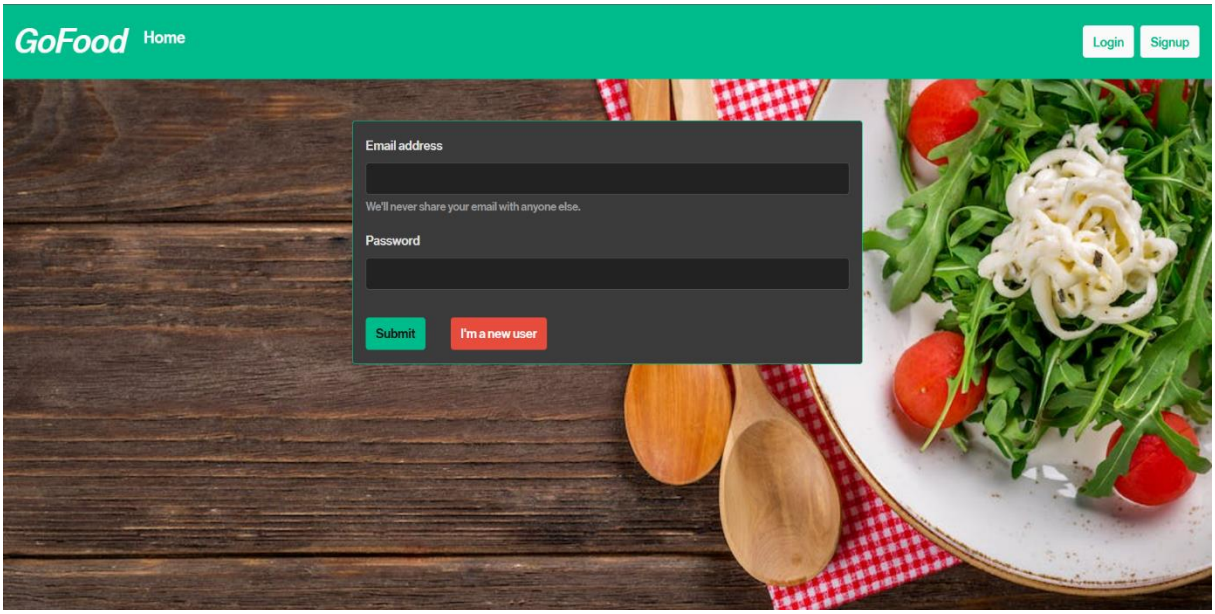


Figure 8.2.3: Login Page

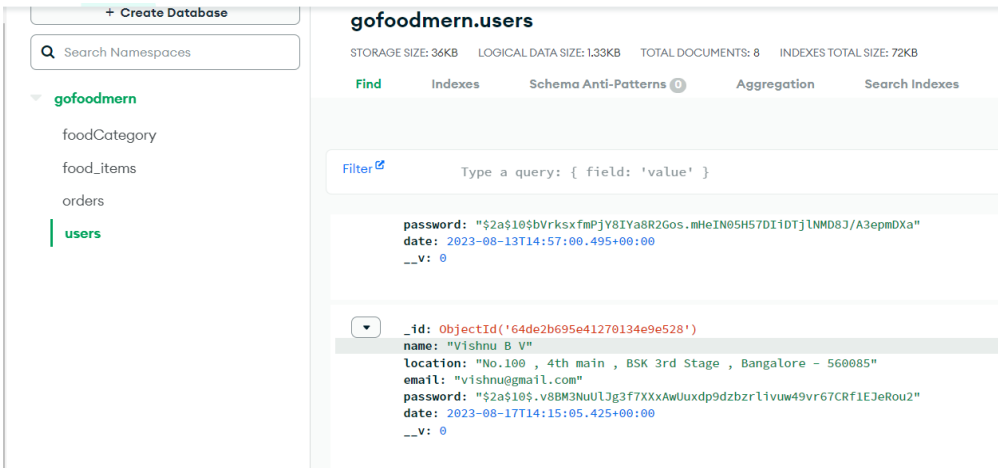


Figure 8.2.4: Storage of Users data in the Database

8.3 Ordering Process

After successfully logging in, users are directed to their personalized account page. Notably, the Navbar undergoes a transformation, showcasing options such as MyOrders, Cart, and Logout, providing enhanced navigation. A convenient search bar is thoughtfully incorporated, enabling users to swiftly locate desired food items by inputting keywords. The food ordering procedure is designed for simplicity. Users can effortlessly customize their orders by specifying quantity and size preferences. Upon completion, a single click on the "Add to Cart" button places the chosen food item into the cart for easy tracking. Moreover, the webpage is thoughtfully optimized for mobile devices, ensuring a seamless browsing experience across various screens. The following images aptly illustrate the intuitive food ordering process.

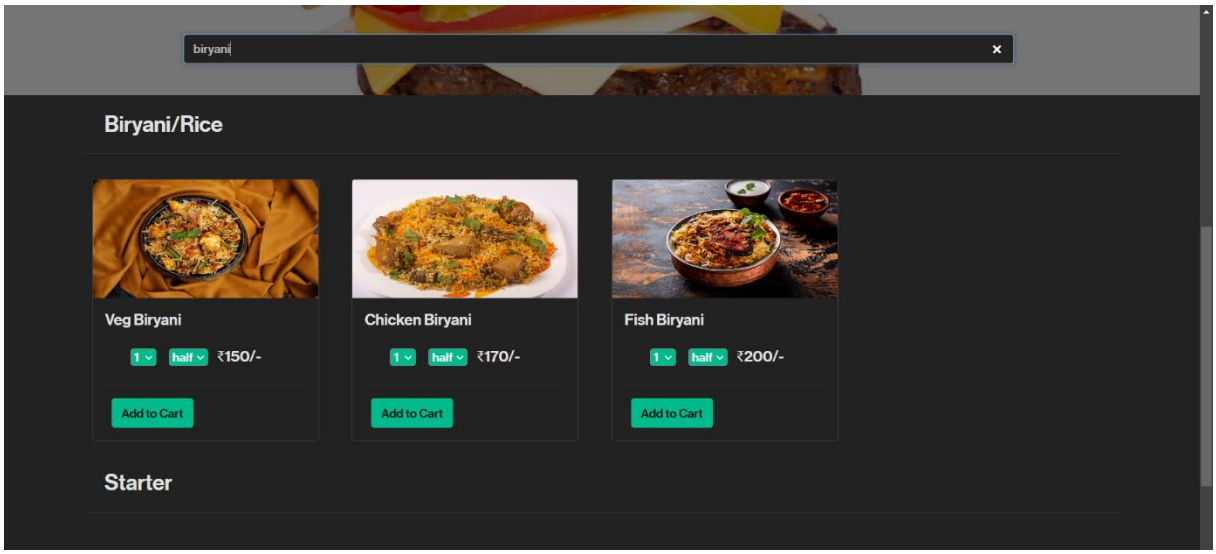


Figure 8.3.1: Search Option

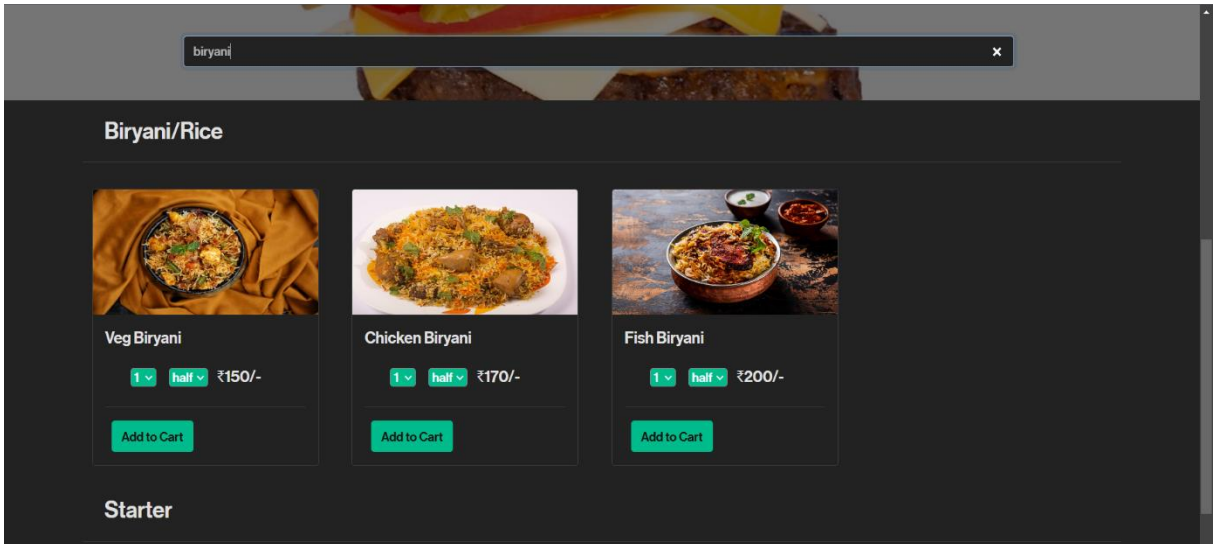


Figure 8.3.2: Ordering Process

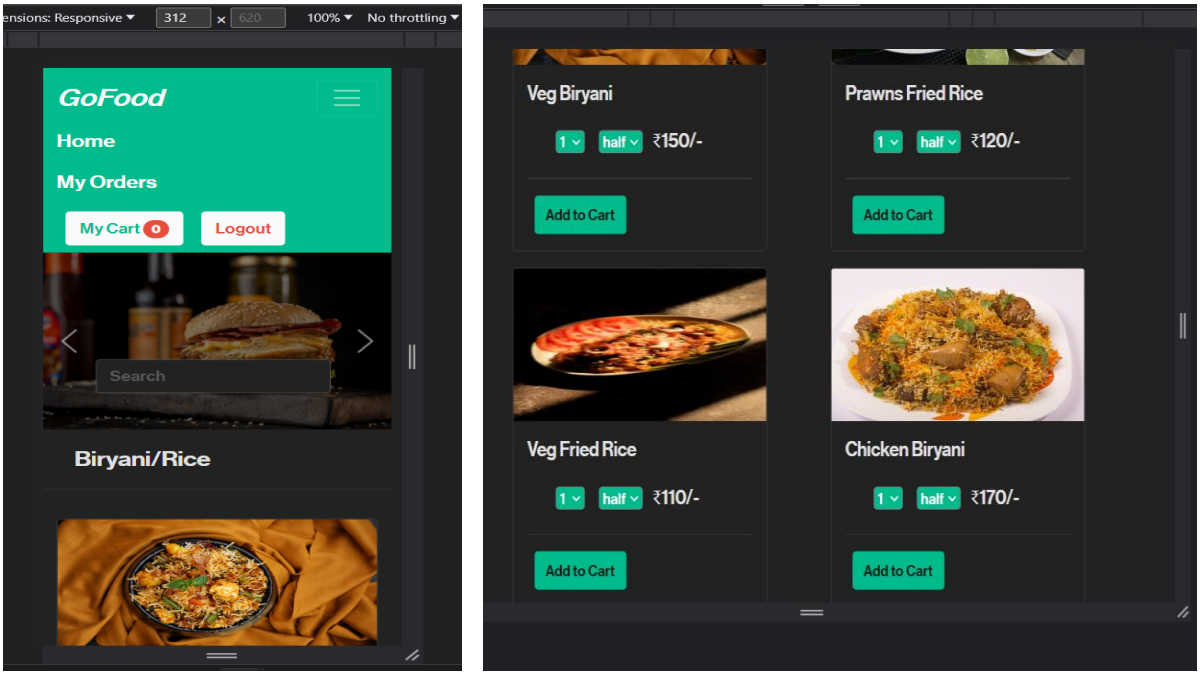


Figure 8.3.3: Responsiveness of the Webapp

8.4 MyCart and MyOrders

Upon selecting food items using the "Add to Cart" button, the chosen items seamlessly populate the "MyCart" section. The badge affixed to the "MyCart" icon dynamically reflects the quantity of items present in the cart; ensuring users are informed at a glance. Within the "MyCart" interface, users are empowered to remove specific food items, fostering a tailored experience. A comprehensive overview is provided, featuring the cumulative order amount and a convenient "Checkout" option. Upon clicking "Checkout," the selected items are promptly updated in the database, facilitating order management. Notably, the ordered items are also accessible via the dedicated "MyOrders" page, allowing users to conveniently track and review their past orders. This streamlined process ensures efficient order placement, modification, and review for a user-friendly experience.

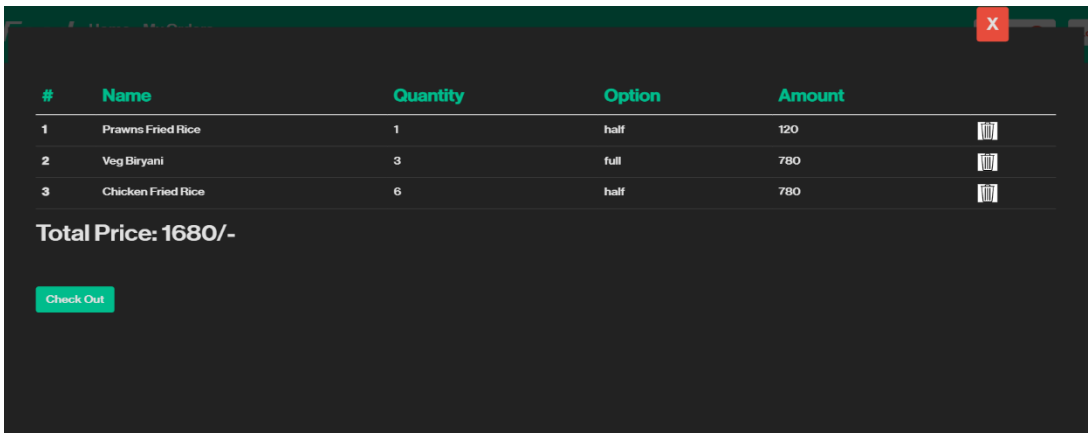


Figure 8.4.1: MyCart Window and Deletion of food option

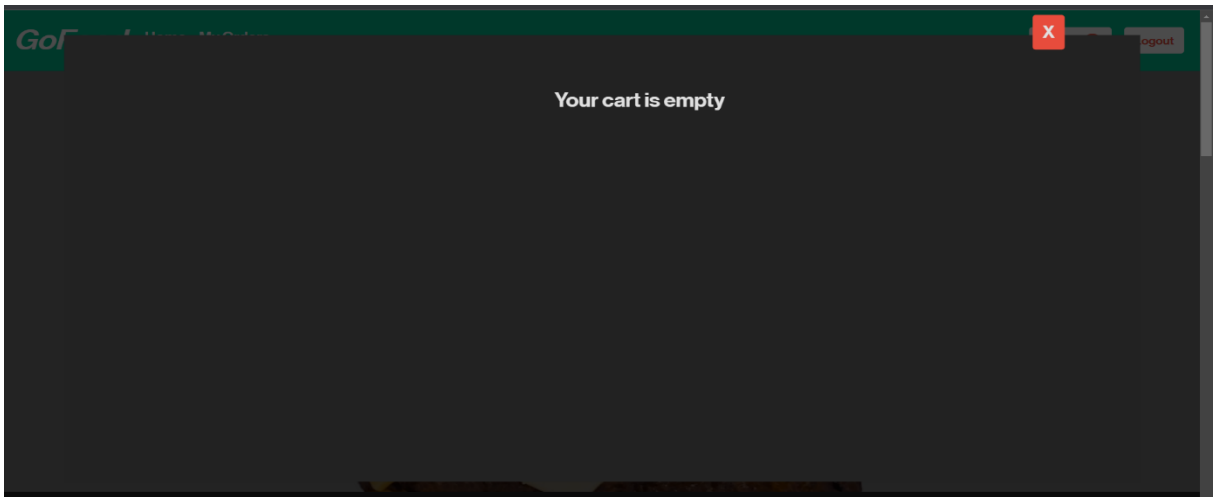


Figure 8.4.2: Once the checkout Option is entered, the Cart is empty and updated

Figure 8.4.3: Database Updating after the checkout process

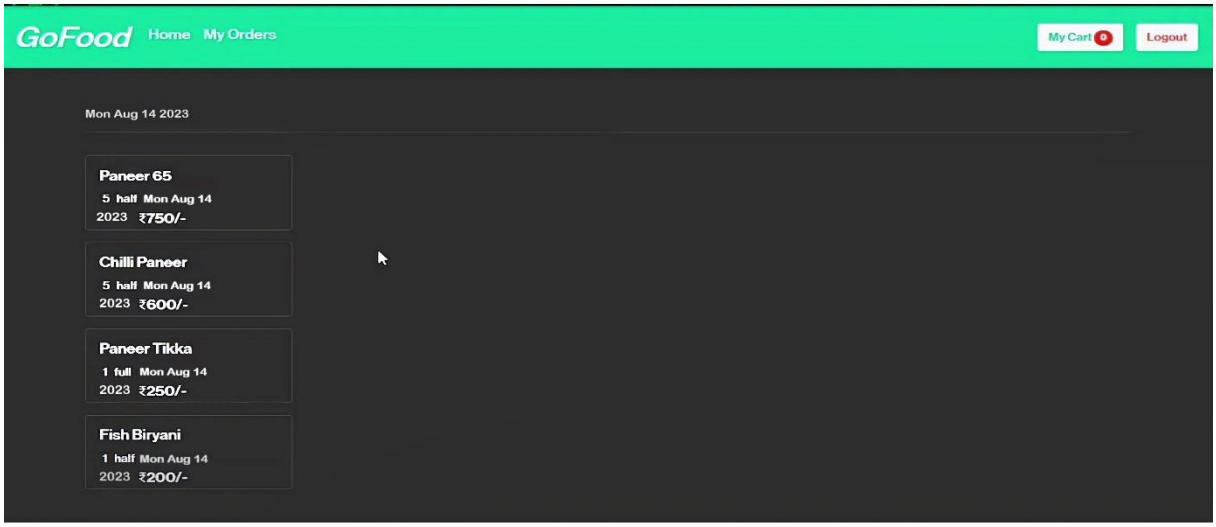


Figure 8.4.4: My Orders Page

CHAPTER- IX

Advantages and Applications

9.1 Advantages

- **Enhanced Convenience:** This feature introduces a convenient way for users to effortlessly place orders and monitor their progress remotely, eliminating the need for physical waiting and streamlining their experience.

- **Elevated User Experience:** By optimizing the ordering process, this enhancement significantly improves user satisfaction. It introduces a seamless, transparent, and engaging journey, fostering positive interactions with the application.

- **Enhanced Order Precision:** Direct user input minimizes the likelihood of misunderstandings or errors during order placement. This accuracy contributes to a smoother order fulfilment process.

- **Crowd Management:** Particularly beneficial for brick-and-mortar establishments like restaurants, the virtual queue feature helps manage foot traffic, supporting social distancing measures in post-pandemic scenarios.

- **Immersive Interface:** Leveraging dynamic and responsive interfaces, the feature immerses users in a visually appealing and interactive environment. This strategy encourages users to actively engage with the app's functionalities.

9.2 Applications

- **Food Ordering:** The central purpose of the application revolves around food ordering. Users can seamlessly explore menus, initiate orders, personalize selections, and monitor the status of their orders in real-time, all without the requirement to be physically present in a queue.

- **Restaurant Management:** Restaurants can use the app to manage incoming orders, update menus, and set opening hours, and monitor customer reviews and feedback.

- **Queue Elimination:** The queue elimination feature is a significant advantage. Users can place orders remotely, reducing the need for physical presence in a queue, which is especially valuable during peak hours or in situations where waiting isn't convenient.

- **Analytics and Insights:** The backend of the app can provide valuable insights to restaurant owners, helping them understand customer behaviour, popular menu items, and peak ordering times.

- **Event Catering:** Attendees at events can order food through the app and receive updates on when and where to pick up their orders.

- **Tailored Marketing for Hotels**: The app offers hotels the advantage of personalized marketing strategies. Hotels can customize the app's branding, menu offerings, and promotional content to align with their unique identity. This customization not only strengthens the hotel's brand presence but also creates a more appealing and engaging experience for their customers, fostering loyalty and repeat business.

CHAPTER- X

Conclusion

The development journey of the "GoFood - A Food Ordering App" has been a remarkable endeavor in creating an efficient and user-centric solution for food enthusiasts. The project's core objective was to establish a seamless platform that simplifies the process of food ordering while providing an enjoyable user experience. By leveraging the MERN (MongoDB, Express, React, Node.js) stack, the project seamlessly translated the initial concept into a fully functional and responsive application.

The inception of the project began with the definition of software requirements, outlining the app's scope, functionalities, and user expectations. This laid the groundwork for a clear development roadmap, addressing challenges and opportunities within the food ordering landscape.

During the implementation phase, the project harnessed the power of React for dynamic user interfaces, Express for robust server-side logic, MongoDB for efficient data management, and Node.js for seamless integration. These technologies worked in harmony to create a cohesive and engaging user journey.

Throughout the development lifecycle, pivotal elements such as user authentication, intuitive navigation, interactive menus, cart management, and responsive design were meticulously integrated to create a holistic user experience. The success of the project lies in the harmonious amalgamation of these components, each contributing to the app's functionality and overall appeal.

In conclusion, the "GoFood - A Food Ordering App" stands as a testament to the project's success in delivering an accessible, user-friendly, and efficient solution for ordering food without waiting in the queue. The integration of cutting-edge technologies and thoughtful design principles has resulted in a platform that not only simplifies the food ordering process but also enhances the overall experience for users.

REFERENCES

[1] https://www.youtube.com/playlist?list=PLI0saxAvhd_OdRWyprSe3Mln37H0u4DAp

[2] https://nodejs.org/en

[3] https://www.npmjs.com/package/nodemon

[4] https://express-validator.github.io/docs/

[5] https://www.npmjs.com/package/bcryptjs

[6] https://jwt.io/

[7] https://account.mongodb.com/account/login?signedOut=true

[8] https://getbootstrap.com/docs/5.0/components/navbar/

[9] https://getbootstrap.com/docs/5.0/components/card/

[10] https://react-bootstrap.netlify.app/

[11] https://reactrouter.com/en/main/components/routes

[12] https://www.istockphoto.com/?http://esource=SEM_IS_GO_IN_Brand_iStock-Mix_EN&kw=CA_iStock-Image_Exact_istock+image_e&kwid=s_43700067636174393_dc&pcrid=562472910442&&&&&gad=1&gclid=CjwKCAjwivemBhBhEiwAJxNWN_Ov9zmxJUXsoPKFCINdeXy5uTGZO3AduzWlYb2gogBY4L0uBc019BoCgkEQAvD_BwE&gclsrc=aw.ds