|  |
| --- |
| Faculty of Applied Sciences and Technology |
| **NODE/EXPRESS WEB API** |
| ITE5315 - Project |
|  |
| **Deep Lad, Dhruv Intwala, Sneh Chavda**  **Submission Date** |
| **4/16/2024** |

|  |
| --- |
| This document explains how to build Node/Express Web API …………………………. |

Table of Contents

[Step 1: Project structure/DB 2](#_Toc153053622)

[Step 2: Routes 3](#_Toc153053623)

[Step 3: FORM/Template Engine 4](#_Toc153053624)

[Step 4: Security 5](#_Toc153053625)

[Step 5: new functionality 6](#_Toc153053626)

[Step 6: deployment 7](#_Toc153053627)

[Bonus questions 8](#_Toc153053628)

[Project planning/sharing tasks 9](#_Toc153053629)

[Summary 10](#_Toc153053630)

# Step 1: Project structure/DB

**index.js:** This file serves as the entry point for your Node.js application. It likely contains the setup for your Express server and routes.

**config/**

* **database.js:** Configuration file for connecting to the MongoDB database. This file contains the necessary settings such as the database URL.

**models/**

* **User.js:** Model file for managing user data. It includes functions for CRUD operations and user authentication (login and registration).
* **Restaurant.js:** Model file for managing restaurant data. Similar to User.js, it handles CRUD operations specific to restaurants.

**views/**

* Contains HBS (Handlebars) templates for rendering views on the client-side. These templates likely include HTML markup with dynamic content.

**Implementation of MongoDB Database:**

* Configuration:
  + In the config/database.js file, I configured the MongoDB connection using the Mongoose library. Mongoose provides a straightforward way to define schemas and interact with the MongoDB database from Node.js.
* Connection Setup:
  + Within the database.js file, I established a connection to the MongoDB database by providing the database URL. This URL typically includes the hostname, port, database name, and authentication credentials.
* Model Definitions:
  + I created separate model files for different data entities such as User and Restaurant in the models/ directory. These model files define the schema structure and methods for interacting with the corresponding MongoDB collections.
* CRUD Operations:
  + Each model file (User.js and Restaurant.js) contains functions for performing CRUD operations (Create, Read, Update, Delete) on the associated MongoDB collections. These functions utilize Mongoose methods such as save, find, findOne, updateOne, and deleteOne to interact with the database.
* Authentication:
  + For user authentication, I implemented login and registration functionality within the User.js model file. This includes hashing passwords, verifying credentials, and generating JWT tokens for authenticated sessions.
* Integration with Express:
  + In the index.js file, I integrated the MongoDB models and routes with the Express application. This involves importing the model files and defining route handlers to handle incoming HTTP requests related to database operations.

# Step 2: Routes

1. **GET /:**
   * Description: Renders the index view.
   * Middleware: verifyToken middleware for token verification.
2. **GET /register:**
   * Description: Renders the user registration form.
   * Middleware: verifyToken middleware for token verification.

A computer screen with a white background

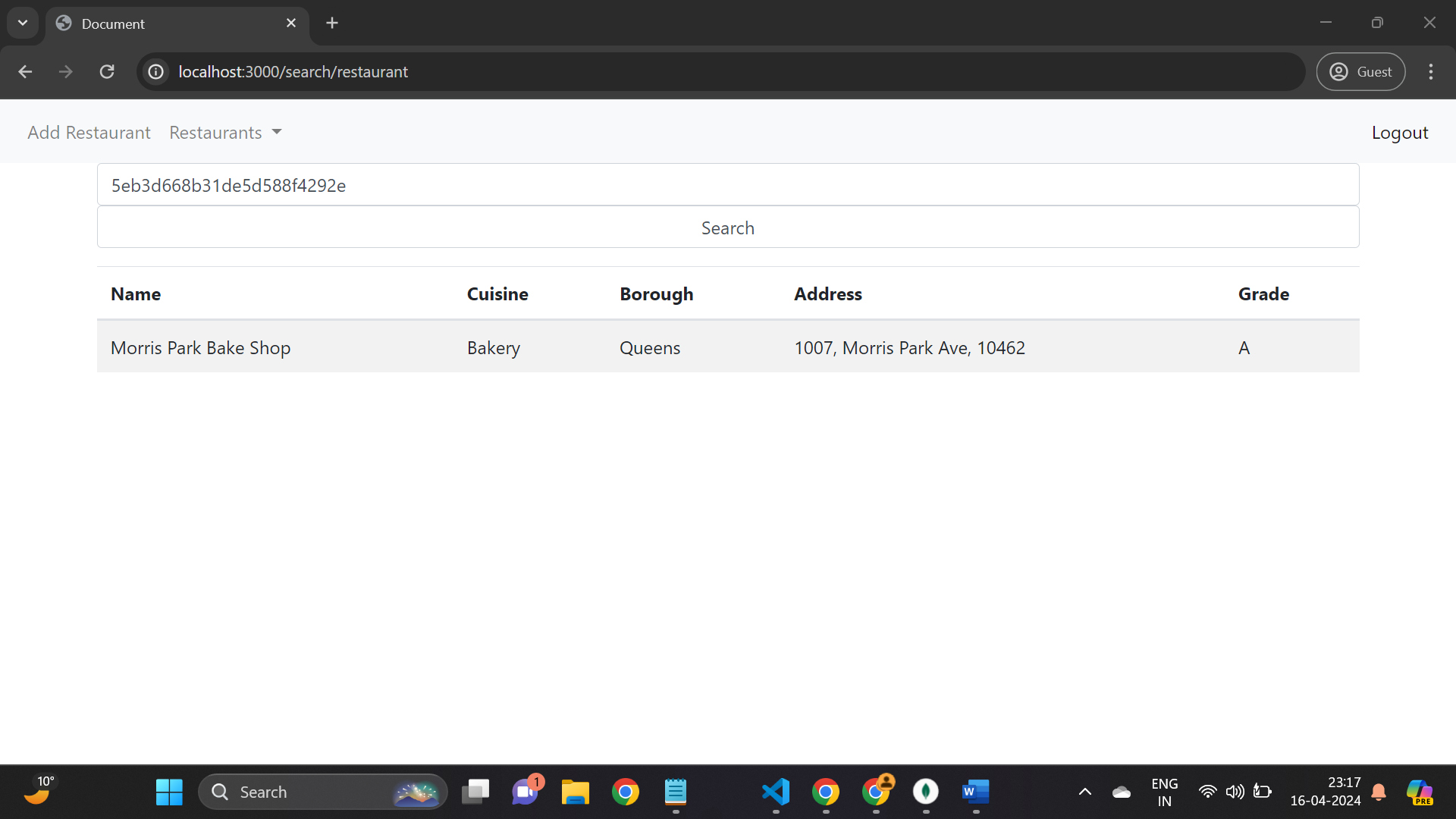
Description automatically generated

1. **POST /register:**
   * Description: Handles user registration.
   * Parameters:
     + email (string): Email address of the user.
     + password (string): Password of the user.
   * Returns: JSON response indicating success or failure.
2. **GET /login:**
   * Description: Renders the login form.
   * Middleware: None.

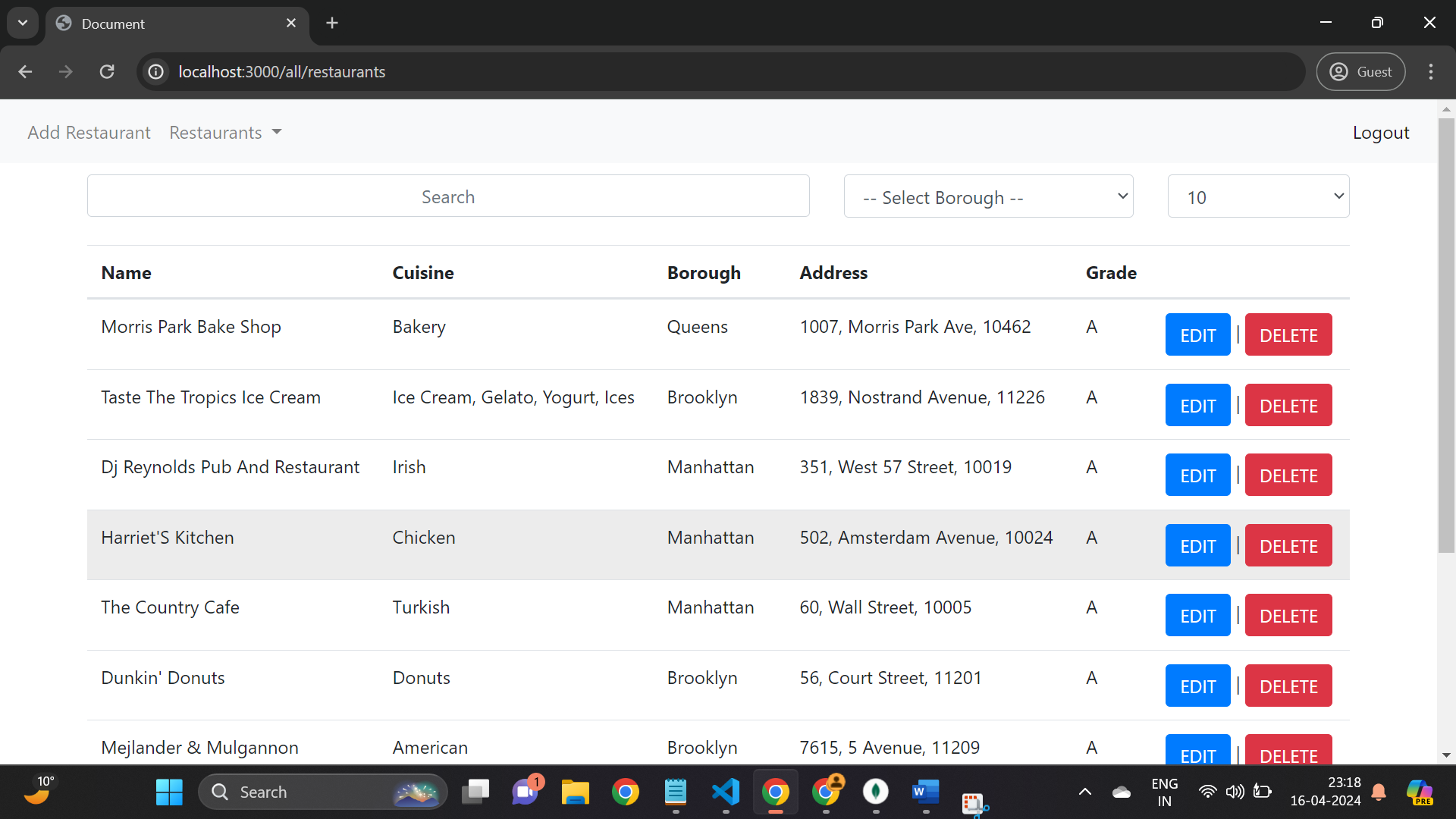
A screenshot of a computer

Description automatically generated

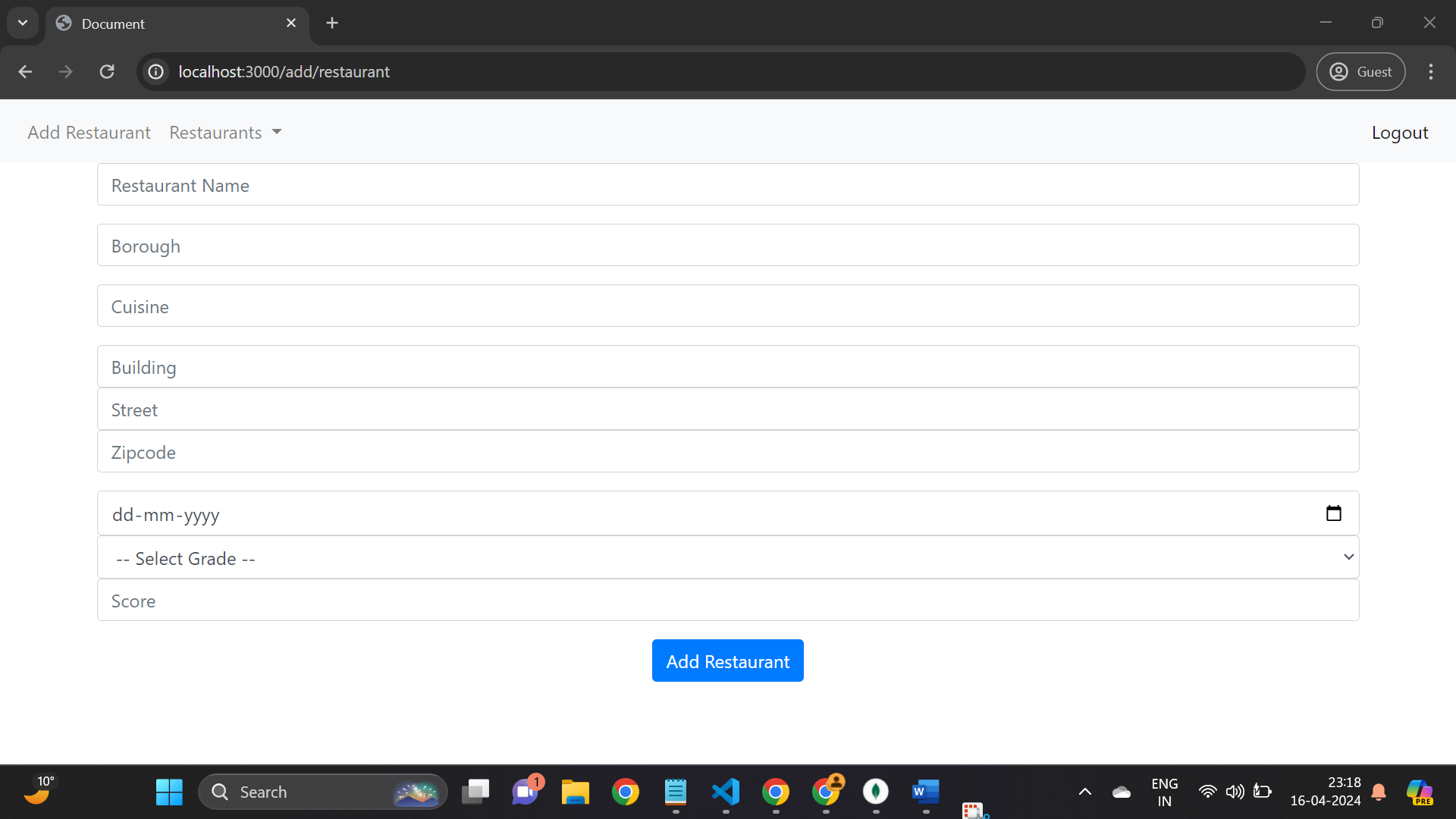
1. **POST /login:**
   * Description: Handles user login.
   * Parameters:
     + email (string): Email address of the user.
     + password (string): Password of the user.
   * Returns: JSON response containing authentication token or error message.
2. **GET /search/restaurant:**
   * Description: Renders the search restaurant view.
   * Middleware: verifyToken middleware for token verification.



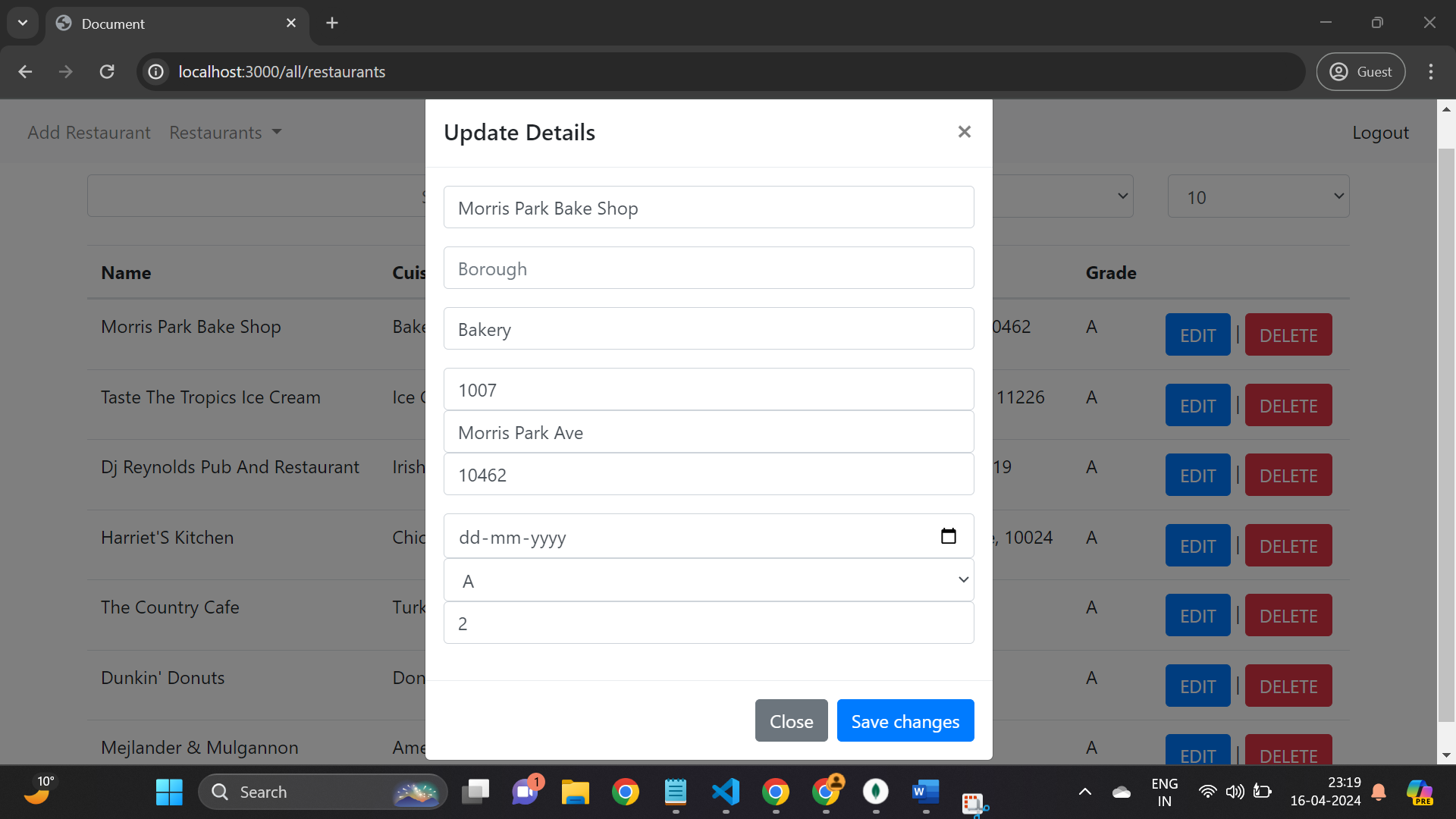
1. **GET /api/restaurants/:id:**
   * Description: Retrieves a restaurant by ID.
   * Parameters:
     + id (string): ID of the restaurant to retrieve.
   * Returns: JSON response containing restaurant details or error message.
2. **GET /all/restaurants:**
   * Description: Renders the view to display all restaurants.
   * Middleware: verifyToken middleware for token verification.



1. **GET /api/restaurants:**
   * Description: Retrieves a list of restaurants.
   * Query Parameters:
     + page (number, optional): Page number for pagination.
     + perPage (number, optional): Number of restaurants per page.
   * Returns: JSON response containing list of restaurants or error message.
2. **GET /add/restaurant:**
   * Description: Renders the add restaurant form.
   * Middleware: verifyToken middleware for token verification.



1. **POST /api/restaurants:**
   * Description: Adds a new restaurant.
   * Parameters:
     + Restaurant details (object): Name, borough, cuisine, address, grades.
   * Returns: JSON response containing newly added restaurant or error message.
2. **PUT /api/restaurants/:id:**
   * Description: Updates a restaurant by ID.
   * Parameters:
     + id (string): ID of the restaurant to update.
     + Updated restaurant details (object): Name, borough, cuisine, address, grades.
   * Returns: JSON response containing success message or error message.



1. **DELETE /api/restaurants/:id:**
   * Description: Deletes a restaurant by ID.
   * Parameters:
     + id (string): ID of the restaurant to delete.
   * Returns: JSON response containing success message or error message.
2. **GET /api/restaurant:**
   * Description: Searches restaurants by query.
   * Query Parameters:
     + query (string): Search query.
   * Returns: JSON response containing search results or error message.
3. **GET /logout:**
   * Description: Logs out the user by clearing the token cookie.
   * Middleware: None.

# Step 3: FORM/Template Engine

**Designing the Form/UI:**

1. **Identify Form Requirements:**
   * Determine the required fields and functionalities for the registration and login forms, considering fields like email and password.
2. **Create Handlebars Templates:**
   * Design Handlebars templates for registration and login forms (**register.hbs** and **login.hbs** respectively). Include HTML markup along with Handlebars expressions for dynamic content rendering.
3. **Implement Form Validation:**
   * Integrate form validation using client-side JavaScript or server-side validation. Ensure validation for fields like email format and password strength.
4. **Handle Form Submissions:**
   * Set up routes (**/add/user** for registration and **/login** for login) to handle form submissions. These routes should validate user input, interact with the User model to perform registration/login operations, and provide appropriate responses.
5. **Customize UI Styling:**
   * Apply CSS styles to enhance the visual appearance and usability of the forms. Customize the appearance of form elements, buttons, and error messages to align with the application's design.

**Testing the Program:**

1. **Manual Testing:**
   * Manually test the registration and login forms by accessing them through a web browser. Verify that the forms render correctly, user input is validated, and form submissions are processed as expected.
2. **Form Validation Testing:**
   * Test form validation by entering invalid data (e.g., incorrect email format, weak password) and verifying that appropriate error messages are displayed to the user.
3. **Backend Testing:**
   * Write unit tests for backend functions related to user registration and login using testing frameworks like Mocha and Chai. Test cases should cover scenarios such as successful registration, login with valid credentials, and error handling for invalid inputs.
4. **Integration Testing:**
   * Perform integration testing to ensure that the forms interact correctly with the backend routes and database operations. Test the entire user registration and login flow, including data validation and authentication processes.

# Step 4: Security

1. **Input Validation:**
   * Utilized the **express-validator** library to validate user input for registration and login forms.
   * Implemented server-side validation to ensure that required fields are not empty and that email addresses and passwords meet specified criteria.
2. **Authentication:**
   * Implemented user authentication using JSON Web Tokens (JWT).
   * Generated JWT tokens upon successful login and stored them as HTTP-only cookies.
   * Created a **verifyToken** middleware function to validate JWT tokens sent with each request and restrict access to protected routes.
3. **Password Security:**
   * Employed secure password handling techniques by hashing user passwords before storing them in the database.
   * Ensured that passwords are not stored in plaintext, enhancing security in case of a data breach.
4. **Cross-Site Request Forgery (CSRF) Protection:**
   * Implemented CSRF protection using the **cookie-parser** middleware to parse and verify cookies.
   * Ensured that CSRF tokens are included in forms and validated with each request to prevent unauthorized actions.
5. **Error Handling:**
   * Implemented error handling middleware to catch and handle errors gracefully.
   * Provided meaningful error messages to users to improve usability and security.
6. **Database Security:**
   * Secured the MongoDB database connection by storing sensitive information such as connection strings and credentials in environment variables.
   * Configured network access controls in MongoDB Atlas to restrict access to trusted IP addresses.

# Step 5: new functionality

**Partial Search Functionality for Restaurant Data**

**Overview:**

In response to user feedback and to enhance the user experience, we have implemented a new functionality that allows users to perform partial searches on restaurant data based on four key fields: name, borough, cuisine, and address. This enhancement aims to provide users with more flexibility in finding relevant restaurants and streamlining the search process.

**Implementation Details:**

1. **Integration with MongoDB:**
   * Leveraging the capabilities of MongoDB, we have implemented a search mechanism that utilizes MongoDB's text search feature for partial matching on specified fields.
2. **Search Algorithm:**
   * When a search query is submitted, the application queries the MongoDB database and performs a partial match search on the specified fields (name, borough, cuisine, and address).
   * The search algorithm retrieves restaurants that contain the search query as a substring in any of the specified fields, allowing for partial matches.
3. **Handling Search Requests:**
   * We have implemented a new route (**/api/restaurant**) to handle partial search requests for restaurant data.
   * The route accepts a search query parameter (**query**) via a GET request and processes the search query.
4. **Query Execution:**
   * Upon receiving a search query, the application constructs a MongoDB query using the **$regex** operator to perform partial matching on the specified fields.
   * The query is executed against the restaurant collection in the MongoDB database, returning matching restaurants based on the search criteria.
5. **Response Format:**
   * The search results are returned to the client as JSON data, containing relevant information about the matched restaurants, such as name, borough, cuisine, and address.
   * If no matching restaurants are found, an appropriate message indicating no records found is returned to the user.

# Step 6: deployment

**Project Deployment on Cyclic:**

* **Hosting Platform:** The Node.js project has been deployed on Cyclic, a cloud hosting platform known for its reliability and scalability.
* **Accessibility:** Deploying the project on Cyclic ensures that it is accessible to users worldwide, providing seamless access to the application's features and functionalities.
* **Scalability:** Leveraging Cyclic's auto-scaling capabilities allows the project to handle varying levels of traffic efficiently, ensuring optimal performance during peak usage periods.
* **Security:** Cyclic offers robust security measures to protect the project from potential threats, including DDoS attacks, data breaches, and unauthorized access.

# Bonus questions

**Utilizing AJAX for Dynamic Data Rendering:**

Our Node.js project incorporates AJAX (Asynchronous JavaScript and XML) to dynamically fetch and render data on the same Handlebars view pages, providing a seamless and interactive user experience. Key points regarding the usage of APIs and AJAX include:

* **API Endpoints:** The application exposes a set of RESTful APIs to interact with the backend server and retrieve data. These APIs are designed to handle various CRUD (Create, Read, Update, Delete) operations for entities such as users, restaurants, and other relevant data.
* **Frontend Integration:** AJAX requests are made from the frontend JavaScript code embedded within the Handlebars view pages. These requests are sent asynchronously to the server to fetch data without requiring a full page reload.
* **Dynamic Rendering:** Upon receiving the API response, the frontend dynamically updates the content of the Handlebars view pages, rendering the retrieved data in the appropriate sections of the page. This enables real-time updates and seamless integration of new data into the existing page structure.
* **Enhanced User Experience:** By leveraging AJAX for data retrieval and rendering, the application provides users with a responsive and interactive interface. Users can view and interact with data on the same page without experiencing interruptions or delays caused by traditional page reloads.
* **Optimized Performance:** AJAX requests minimize network latency and server load by fetching only the necessary data asynchronously. This results in faster page load times and improved overall performance, enhancing the user experience and reducing bandwidth consumption.
* **Error Handling:** The frontend code includes robust error handling mechanisms to gracefully handle API errors and network failures. This ensures that users receive meaningful error messages and prompts for action in case of any issues encountered during data retrieval.

# Project planning/sharing tasks

(Describe how did you divide the work and did project planning. Have you changed the day-1-planning/milstones/deliverables that we did at the beginning of project in the class?)

* We tried to evenly divide the tasks.
* Login, registration and logout pages along with the backend logic was done by Sneh Chavda.
* Dhruv was responsible for implementing add restaurant and search by id functionality.
* Deep was responsible for implementing get all restaurants, partial search, and pagination.
* Implementation of JWT was collectively done by all three members of the group as it consisted of more complex problem solving.

# Summary

Our Node.js project is a comprehensive web application designed to facilitate user registration, authentication, restaurant management, and dynamic data rendering using AJAX. The application leverages a MongoDB database for data storage and retrieval, while the frontend is built using Handlebars for templating and AJAX for asynchronous data fetching and rendering.

**Implementation Details:**

1. **Backend Development:**
   * The backend of the application is developed using Node.js and Express.js, providing a robust server-side framework for handling HTTP requests and responses.
   * MongoDB is utilized as the database management system, with Mongoose serving as the ODM (Object-Document Mapper) to interact with MongoDB from Node.js.
   * RESTful APIs are implemented to enable CRUD operations for entities such as users and restaurants, allowing for seamless data manipulation and management.
2. **Frontend Development:**
   * Handlebars is used as the templating engine for generating dynamic HTML content on the client-side, enabling the rendering of server-side data within the views.
   * AJAX (Asynchronous JavaScript and XML) is employed to fetch data from the server asynchronously, enhancing the user experience by providing real-time updates without full page reloads.
3. **Authentication and Security:**
   * User authentication is implemented using JSON Web Tokens (JWT), providing a secure mechanism for managing user sessions and ensuring authenticated access to protected routes.
   * Input validation and error handling mechanisms are integrated into both the frontend and backend to enhance security.
4. **Deployment:**
   * The project is deployed on Cyclic, a reliable and scalable cloud hosting platform, ensuring high availability, scalability, and security for the application.
   * Cyclic's auto-scaling capabilities and security features contribute to the optimal performance and reliability of the deployed application.