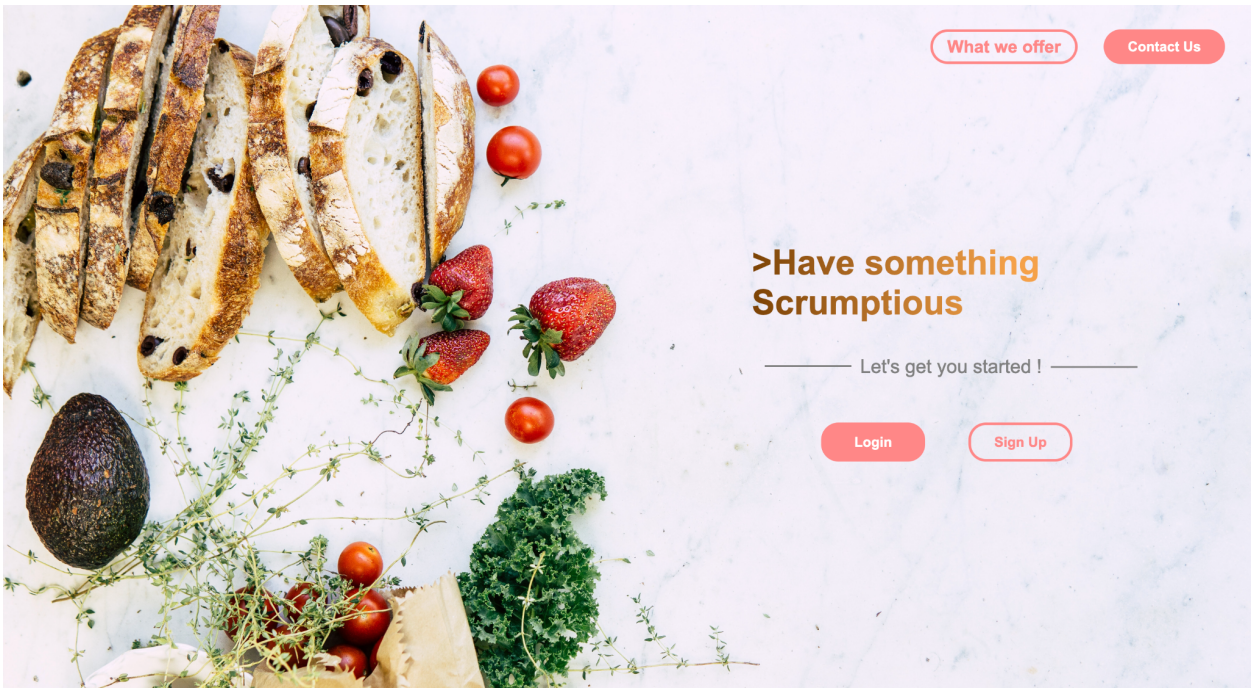
Food Ordering Application

I have developed a comprehensive food ordering application tailored specifically for office staff using a combination of HTML, CSS, JavaScript, Node.js, MongoDB, and EJS. Leveraging the power of these technologies, I have successfully deployed the application using render. The application encompasses a range of functionalities to enhance user experience. Upon registration, users are required to fill in their details, ensuring a personalized experience. Users can conveniently browse through an extensive menu, select their desired dishes, and place orders effortlessly. This food ordering application is designed to streamline the ordering process for office staff, providing a seamless and user-friendly experience.

Homepage.ejs: Landing Page for my food ordering application



In the food ordering application I have developed, **I have implemented the Model-View-Controller (MVC) architectural pattern**.

MVC is a software design pattern that separates the application into three interconnected components: **the Model, the View, and the Controller**.

- **Model:** The Model represents the data and business logic of the application. In the food ordering application, the Model component handles tasks such as interacting with the database (MongoDB in this case), performing data validation, and executing business

operations. It encapsulates the essential functionalities related to user details, dishes, and orders. The Model component provides an abstraction layer that isolates the underlying data from the rest of the application.

- **dishesModel.js**: This file establishes a connection to a MongoDB database using Mongoose, defines the schema for the "dishes" collection, and exports a Mongoose model for performing database operations related to dishes.
  - The schema includes fields such as "name", "Amount", "Image", "Details", and "category", each with its respective data type and optional validation rules.
  - Creates a Mongoose model using mongoose.model(), which maps the "dishesSchema" to the "dishes" collection in the database.
- **pendingOrderModel.js:** This file establishes a connection to a MongoDB database using Mongoose, defines the schema for the "orders" collection specifically for pending orders, and exports a Mongoose model for performing database operations related to pending orders.
  - The schema includes fields such as "name", "amount", "address", and "dishes", each with its respective data type and required validation rules.
  - Creates a Mongoose model using mongoose.model(), which maps the "OrderSchema" to the "orders" collection in the database.
- **usersModel.js:** this file establishes a connection to a MongoDB database using Mongoose, defines the schema for the "users" collection, and exports a Mongoose model for performing database operations related to user data.
  - The schema includes fields such as "name", "email", "password", "confirmPassword", "role", "pImage", "phone", "orgName", "eid", and "eidpic", each with its respective data type and validation rules.
  - Additionally, the schema includes a pre-save middleware function using userSchema.pre() to modify the document before it is saved. In this case, it sets the confirmPassword field to undefined, ensuring it is not stored in the database.

- **View:** The View component is responsible for presenting the user interface to the application's users. In the food ordering application, the View displays the various screens and forms where users can view menus, select dishes, provide their details, and

place orders. It ensures a visually appealing and intuitive interface that allows users to interact with the application effortlessly.

    - In the "view" folder of my project, I have HTML files that were later converted to EJS templates. These HTML files contain the structure and layout of the different views or pages in the application. They are typically associated with specific routes or functionality.

    - Initially, the HTML files were likely standalone files with their respective CSS and JavaScript files linked within the HTML using <link> and <script> tags. However, after converting them to EJS templates, the file extension was changed to .ejs, indicating that they now support embedded JavaScript code.

    - In EJS templates, I have included dynamic content and logic by embedding JavaScript code within <% %> tags. This allows us to dynamically render data, generate HTML content, and make decisions based on the server-side data.

    - Overall, the "view" folder contains the converted EJS templates that define the structure of the application's pages, and we can include CSS and JavaScript files within these templates to enhance their styling and functionality.

- **Controller:** The Controller acts as an intermediary between the Model and the View. It handles user interactions, processes requests, and updates the Model and View accordingly. In the food ordering application, the Controller receives user input, validates it, and triggers appropriate actions. For example, when a user selects dishes and places an order, the Controller manages the interaction with the Model to save the order details in the database and updates the View to reflect the successful order placement.

    - **authController.js:** This file contains a set of functions that handle user authentication and authorization in the application. Here is a concise summary of what each function does:

        - **Signup Function:** Handles the user signup process. It receives user data from the request body such as username, email, password, confirmPassword, phone, org, and eid. It creates a new user using the userModel, storing the user details in the database. It generates a JSON Web Token (JWT) containing the user's ID and signs it with a secret key. The JWT is then set as a cookie in the response. Finally, it sends a response indicating a successful signup with the newly created user data.

        - **Login Function:** Handles the user login process. It receives the user's email and password from the request body. It searches for a user with the provided email in the userModel. If a user is found, it compares the

provided password with the stored password. If they match, it generates a JWT and sets it as a cookie in the response. It sends a response indicating a successful login with the user data. If the email or password is incorrect, appropriate error messages are sent in the response.

- **Logout Function**: Clears the JWT cookie from the response, effectively logging the user out. It sends a response indicating a successful logout.

- **isLoggedIn Middleware:** Checks whether a user is logged in or not. It verifies the JWT from the cookie in the request. If the token is valid, it retrieves the user data from the userModel based on the token payload and adds the user data to the request object. This middleware is used to authenticate users for protected routes.

- **protectRoute Middleware**: Protects a route by checking if the user is logged in. It verifies the JWT from the cookie in the request. If the token is valid, it adds the user's ID to the request object and allows the request to proceed to the next middleware or route handler. If the token is invalid or not provided, appropriate error messages are sent in the response.

- **dishController.js:** This file contains three functions that handle the operations related to dishes in the application. Here is a concise summary of what each function does:

  - **getAllDishes:** Retrieves all dishes from the dishesModel by calling the find method with an empty query object. It returns a response with a status code of 200 and a JSON object containing a success message and the retrieved dishes.

  - **deleteDish:** Deletes a dish based on the provided dish ID. It receives the dish ID from the request body. It calls the deleteOne method on the dishesModel, passing the dish ID as the query.

  - **addDish:** Adds a new dish to the database. It receives the dish details such as name, amount, image, details, and category from the request body. It also receives the dish image file through a file upload. It creates a new dish document using the dishesModel, storing the dish details and the image path in the database.

- **orderController.js:** This file contains four functions related to order management in the application. Here is a summary of what each function does:

  - **getAllOrder:** Retrieves all orders from the OrdereModel by calling the find method with an empty query object. It returns a response with a

status code of 200 and a JSON object containing a success message and the retrieved orders.

- **deleteOrder:** Deletes an order based on the provided order ID. It receives the order ID from the request body. It calls the deleteOne method on the OrdereModel, passing the order ID as the query.

- **addOrder:** Adds a new order to the database. It receives the order details such as name, amount, address, and dishes from the request body. It creates a new order document using the OrdereModel, storing the order details in the database. It also clears the "cart" cookie.

- **addcart:** Adds a "cart" cookie to the response. It receives the user ID and cart details from the request body. It generates a JSON Web Token (JWT) using the jwt.sign method, including the user ID and cart data. It sets the "cart" cookie in the response with the generated token.

- **userController.js:** This file exports a single function called "updateProfilePhoto" that handles the updating of a user's profile photo. The function receives a file from the request, saves it to the appropriate destination, and retrieves the user's ID from the request. It then finds the corresponding user in the userModel using the ID, updates the user's profile photo path with the newly uploaded image path, and saves the changes to the user's document in the database.

- **viewController.js:** The file defines functions for handling various routes, such as the login page, signup page, menu page, profile page, order history page, homepage, FAQ page, contact page, change menu page, and cart page. These functions check if the user is authenticated, and based on that, they either render the respective views or redirect the user to the appropriate page. The functions interact with models such as dishesModel and OrdereModel to retrieve data and pass it to the views for rendering. Additionally, the "cart" function handles the "cart" cookie, verifies its contents using JWT, calculates the total amount, and renders the cart view with the relevant data. Overall, this file manages the rendering of different views and handles user authentication and data retrieval for those views in the application.

By implementing the MVC architecture in the food ordering application, the codebase is divided into three distinct components, promoting separation of concerns, code reusability, and maintainability.

Let’s dive deep and understand what other files are doing?

Public Folder: The "public" folder in the project contains all the static assets such as CSS files, JavaScript files, and images. These assets are publicly accessible by the clients and can be

referenced and loaded into the HTML templates or views. The CSS files define the styles and layout of the web pages, while the JavaScript files provide interactive functionality and behavior. The images stored in the "public" folder can be referenced in the HTML and CSS code using relative paths to enhance the visual content of the website.

App.js: Let's start with the main file app.js, It sets up the server, configures middleware, establishes routing paths, and starts the server to handle incoming requests for the food ordering application.

- Imports the required dependencies, including Express, path, and cookie-parser.
- Creates an instance of the Express application.
- Set up the middleware to handle JSON data and cookies.
- Serves static files from the "public" directory.
- Sets the view engine as EJS and specifies the directory where the view templates are located.
- Defines the routes for different functionality areas such as user management, dishes, orders, and views.
- Specifies the base URL paths ("/user", "/dishes", "/order") for routing requests to the respective routers.
- Starts the server and listens on the specified port, either from the environment variable "PORT" or defaulting to 3000.

Build.js: The build.js script automates the conversion of an EJS template file to an HTML file by using the fs and ejs modules. It provides a convenient way to generate static HTML files from dynamic EJS templates.