### **Title: Displaying all Places by UserID from MongoDB Atlas collection(Assignment-9)**

**Introduction:**

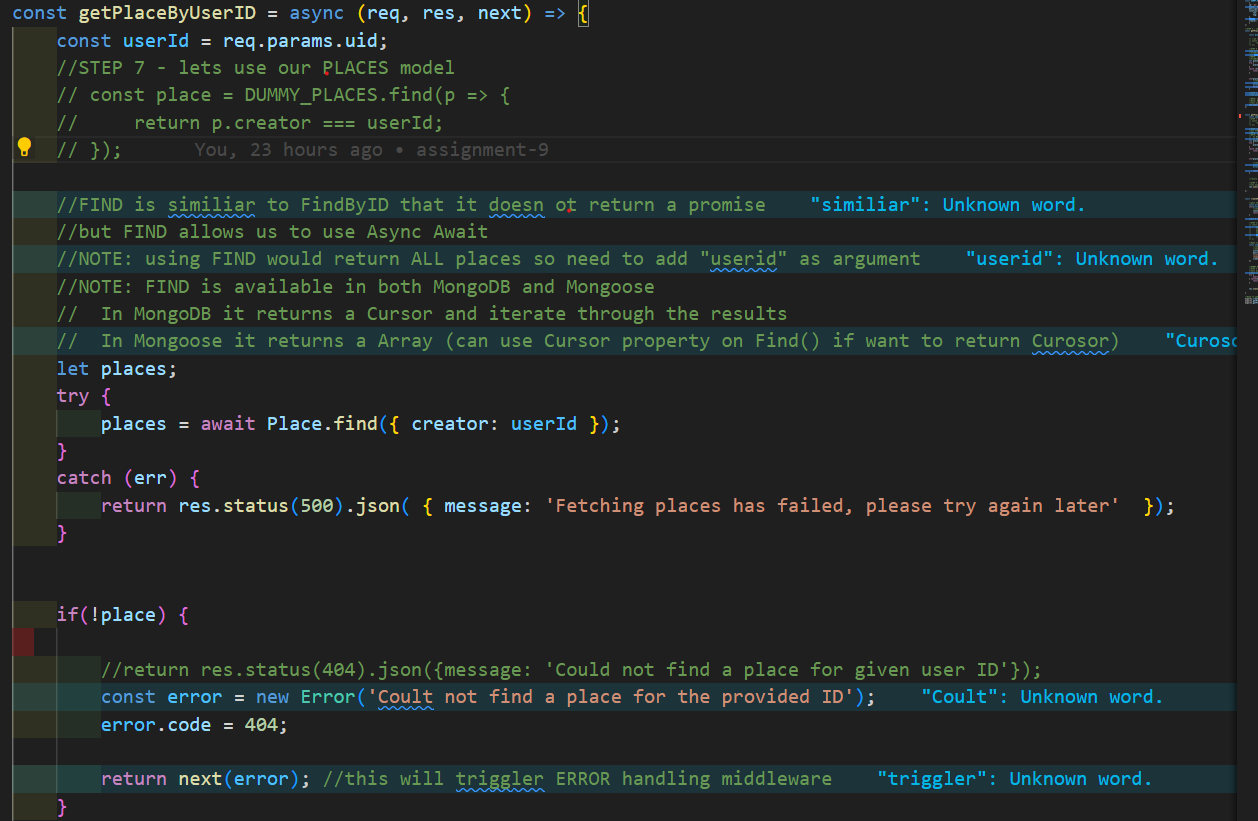
In this project, we have integrated and created all placed API and fetched data form MongoDB Atlas.

**Part One:**

**Step: 1: Backend Code Modification**

Modified getPlaceByUserID function as per the requirements:

The getPlaceByUserID function is an asynchronous function designed to fetch and return all places created by a specific user, identified by their user ID.



Description:

Extract User ID:

The user ID is extracted from the URL parameters (req.params.uid).

Database Query:

The function attempts to find all places in the database where the creator field matches the provided user ID using Place.find({ creator: userId }).

Error Handling:

If the database query fails, a 500 status code is returned with a message indicating the failure.

If no places are found, or the query returns an empty array, a 404 error is created and passed to the next middleware function.

Response:

If places are found, they are mapped to plain JavaScript objects using toObject with the getters option set to true.

The resulting array of places is then sent back in the response.

Example Usage:

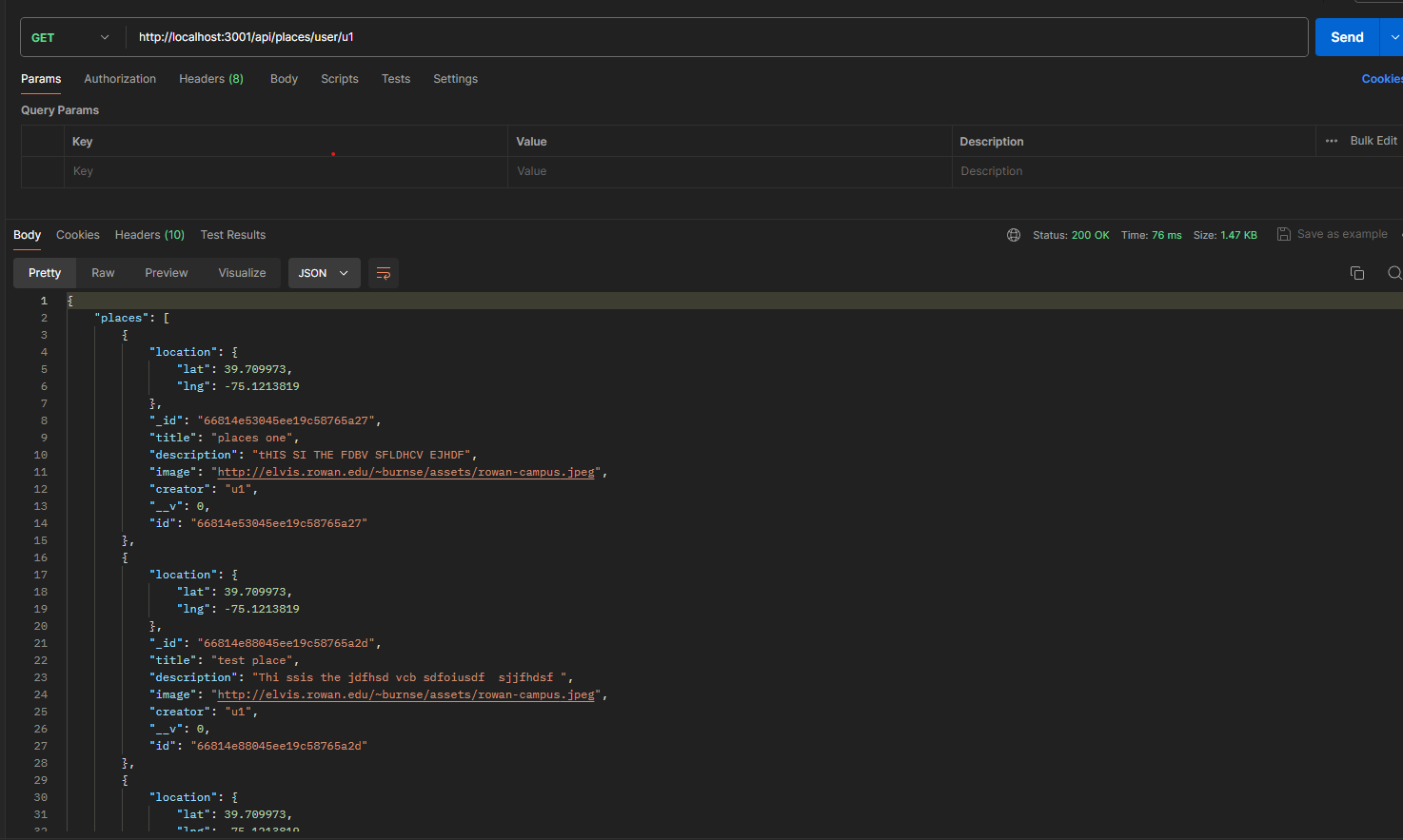
URL: /api/places/user/:uid

Method: GET

Description: This endpoint retrieves all places created by the user with the specified user ID.

Postman Screeenshot:

To verify the functionality of the getPlaceByUserID endpoint, a Postman request can be made to simulate a client request. Below is a description of how to set up and interpret the Postman request for this endpoint.



Postman Setup Instructions:

URL:

Set the request URL to http://<your-server-address>/api/places/user/:uid

Replace <your-server-address> with your actual server address.

Replace :uid with the specific user ID you want to query.

Method:

Select GET as the HTTP method.

Headers:

Ensure that the required headers are set, such as Content-Type if needed. Typically, for a GET request, no additional headers are required.

Authorization:

If your endpoint requires authorization, add the appropriate headers or tokens under the "Authorization" tab.

Example Request:

URL: http://localhost:5000/api/places/user/123

Method: GET

Example Response:

Status Code: 200 OK

**Step 2: Frontend Intigration:**

A: UserPlaces Component:

Purpose::

The UserPlaces component is designed to fetch and display a list of places created by a specific user. It uses a custom HTTP hook (useHttpClient) to send requests to a backend API and manages the fetched data using React state.

Import Necessary Modules:

useEffect and useState from React for state management and side effects.

useParams from react-router-dom to extract route parameters.

PlaceList component to display the list of places.

useHttpClient custom hook for sending HTTP requests.

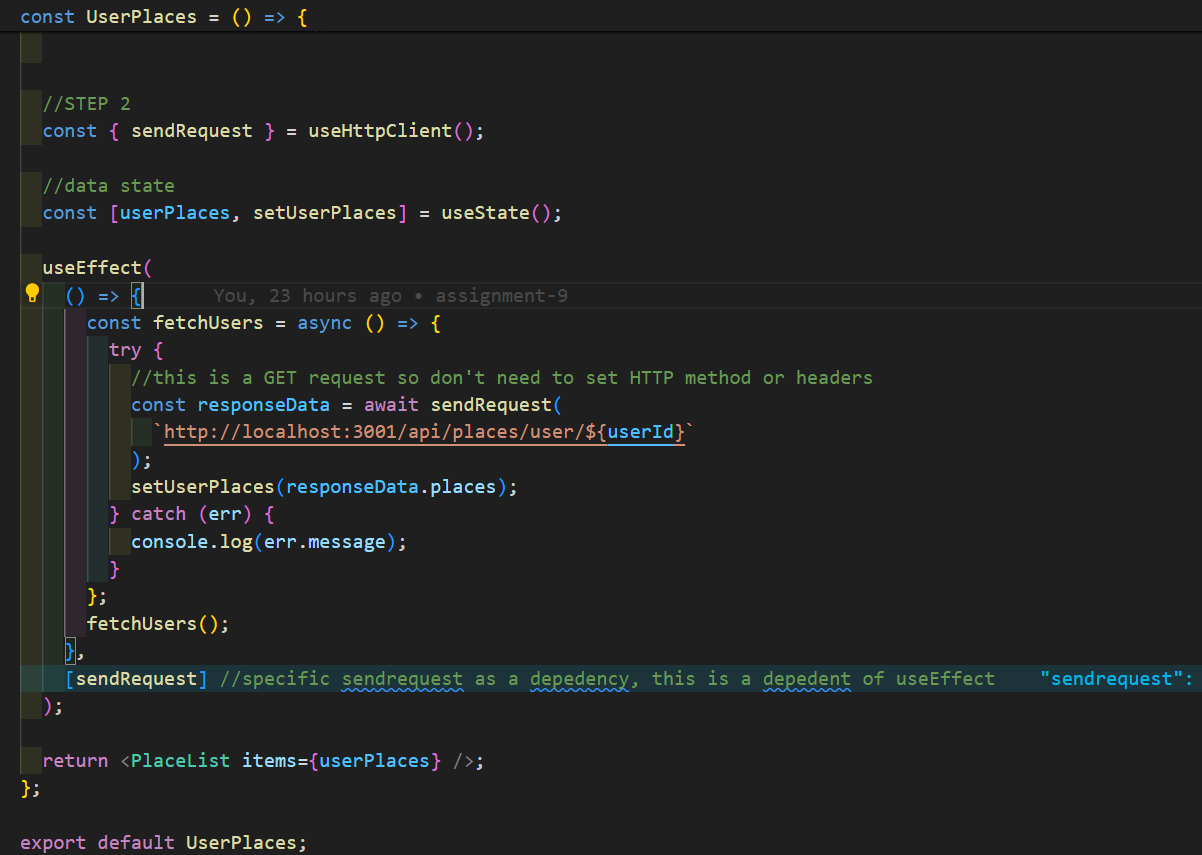
Fetching Data:

The useEffect hook is used to fetch the user's places when the component mounts.

The sendRequest function sends a GET request to the backend API to fetch places for the specified user.

The fetched data is stored in the userPlaces state.

Screenshot of a code:



B: PlaceList Component

Purpose:

The PlaceList component is designed to display a list of places. If no places are available, it shows a message indicating that there are no places for the current user and provides a button to create a new place.

Import Necessary Modules:

Importing Card and Button components from shared components.

Importing PlaceItem to render each place.

Importing PlaceList.css for styling.

Props:

items: An array of place objects. Each place object contains details such as id, image, title, description, address, creator, and location.

Styling:

The component uses CSS classes place-list and center for styling. The styles are imported from the PlaceList.css file.

Functionality:

No Places Message:

If the items array is empty or undefined, the component renders a message indicating that there are no places for the current user.

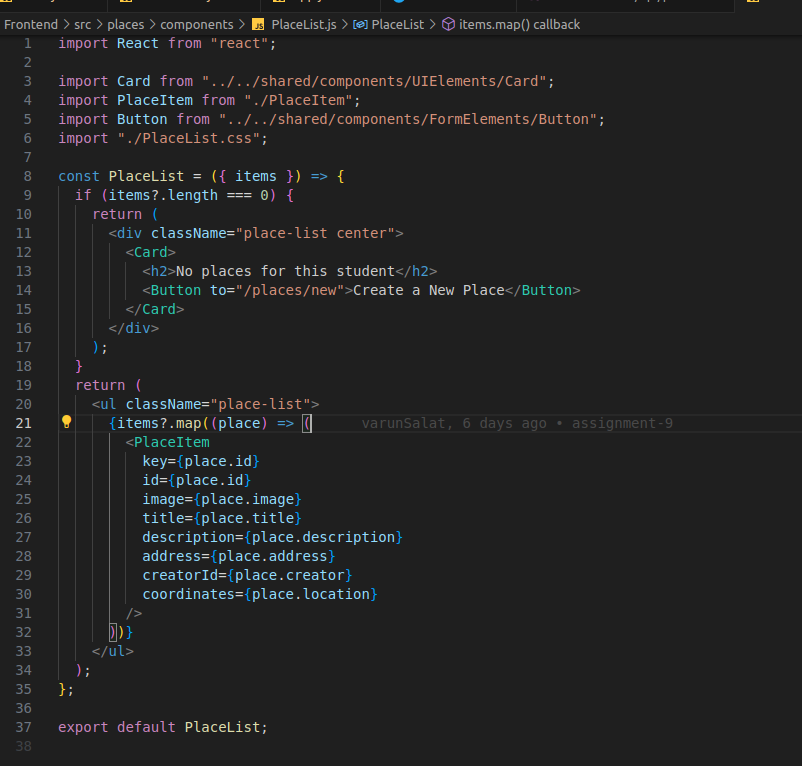
A button is provided to navigate to the page for creating a new place.

Places List:

If the items array contains place objects, the component maps over the array and renders a PlaceItem component for each place.

Each PlaceItem is passed props corresponding to the details of each place.

Screenshot of Places List Component:



C: PlaceItem Component:

Purpose:

The PlaceItem component is designed to display the details of a single place. It includes functionality for editing and deleting the place, with a confirmation modal for deletions.

Styling:

The styles are imported from the PlaceItem.css file.

Props:

id: The unique identifier of the place.

image: The URL of the place's image.

title: The title of the place.

description: A brief description of the place.

address: The address of the place.

creatorId: The ID of the user who created the place.

coordinates: The geographical coordinates of the place.

State:

showConfirmModal: A boolean state to control the visibility of the delete confirmation modal.

Context:

AuthContext: Used to check if the user is logged in.

Functionality:

Delete Confirmation Modal:

showDeleteWarningHandler: Sets showConfirmModal to true, displaying the delete confirmation modal.

cancelDeleteHandler: Sets showConfirmModal to false, hiding the delete confirmation modal.

confirmDeleteHandler: Logs "DELETING..." to the console and hides the delete confirmation modal.

Rendering:

Displays place details, including image, title, address, and description.

If the user is logged in, shows "EDIT" and "DELETE" buttons.

Clicking "DELETE" triggers the delete confirmation modal.

The modal includes "CANCEL" and "DELETE" buttons for user confirmation.

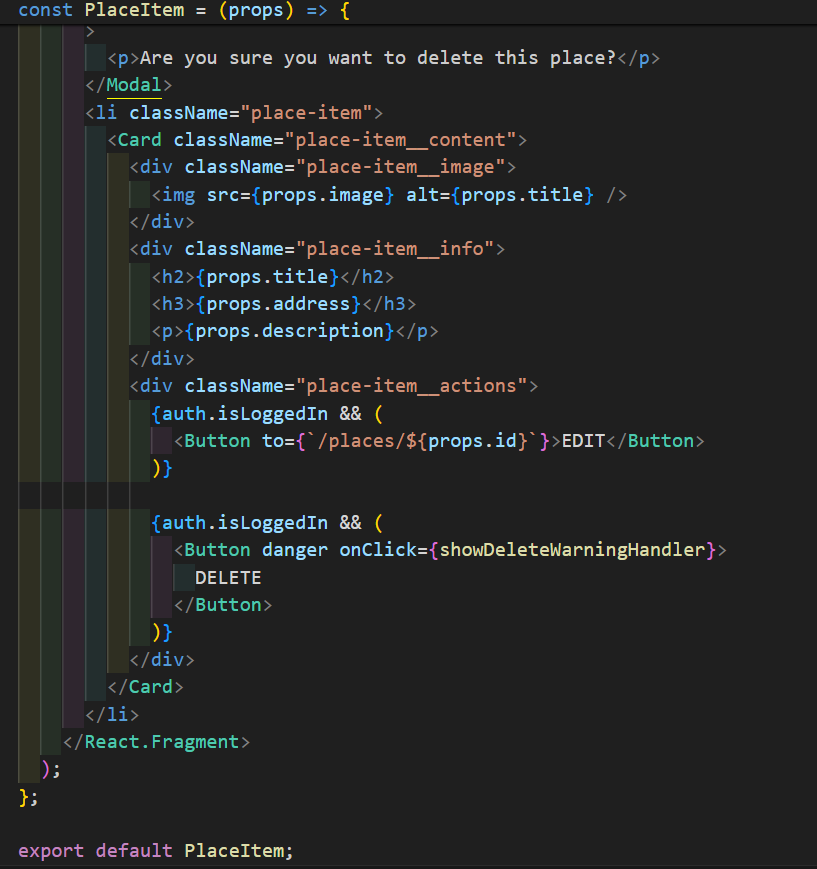
Child Components:

Card: A UI component to wrap the place content.

Button: A component for navigation and actions (edit, delete).

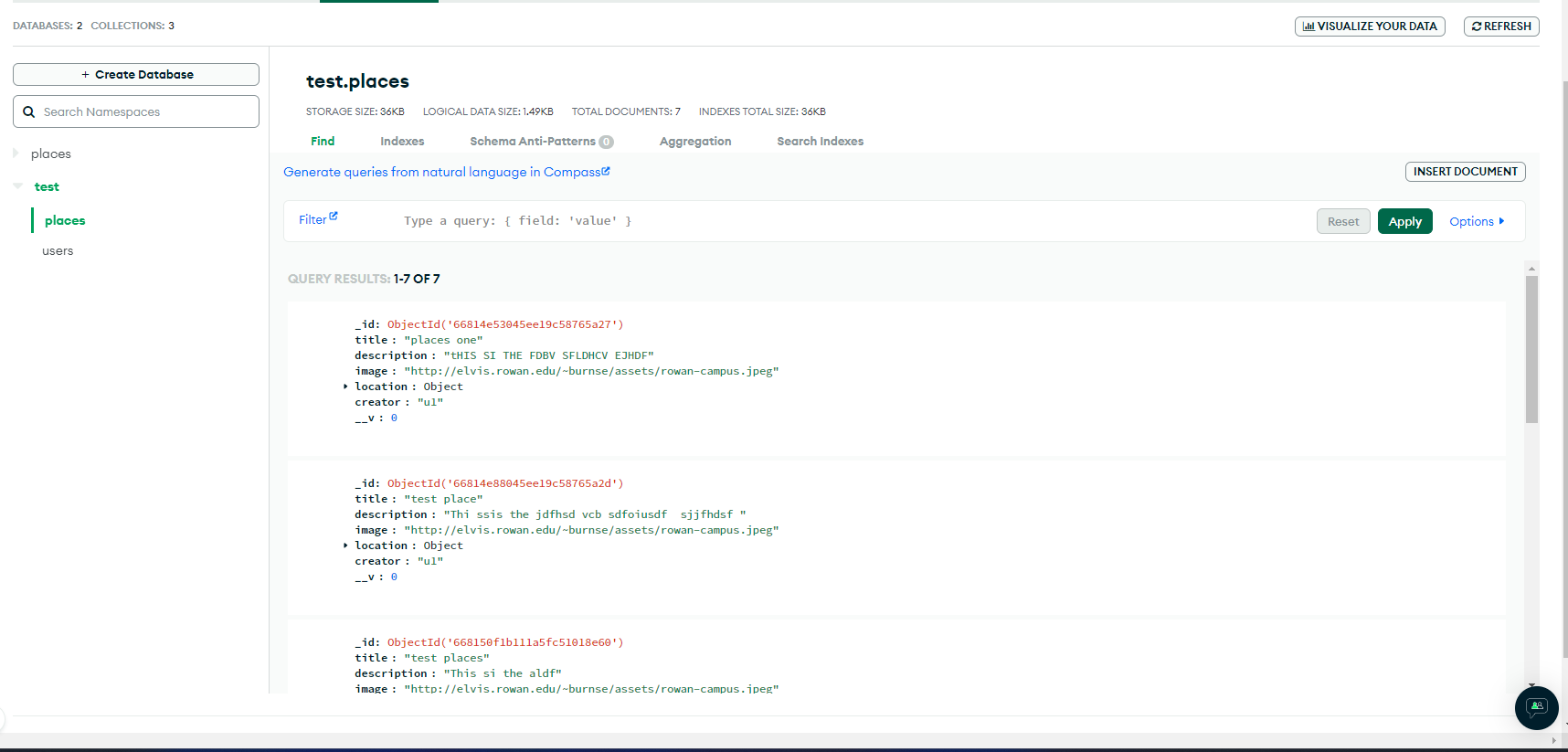
Modal: A component for displaying the delete confirmation modal.

Screenshot of Places PlaceItem Component:



**MongoDB Atlas Screenshot:**

To provide a clear understanding of the data structure stored in MongoDB, a screenshot of the place data is attached. This screenshot showcases the documents in the "places" collection, highlighting the fields and data types used for each place.

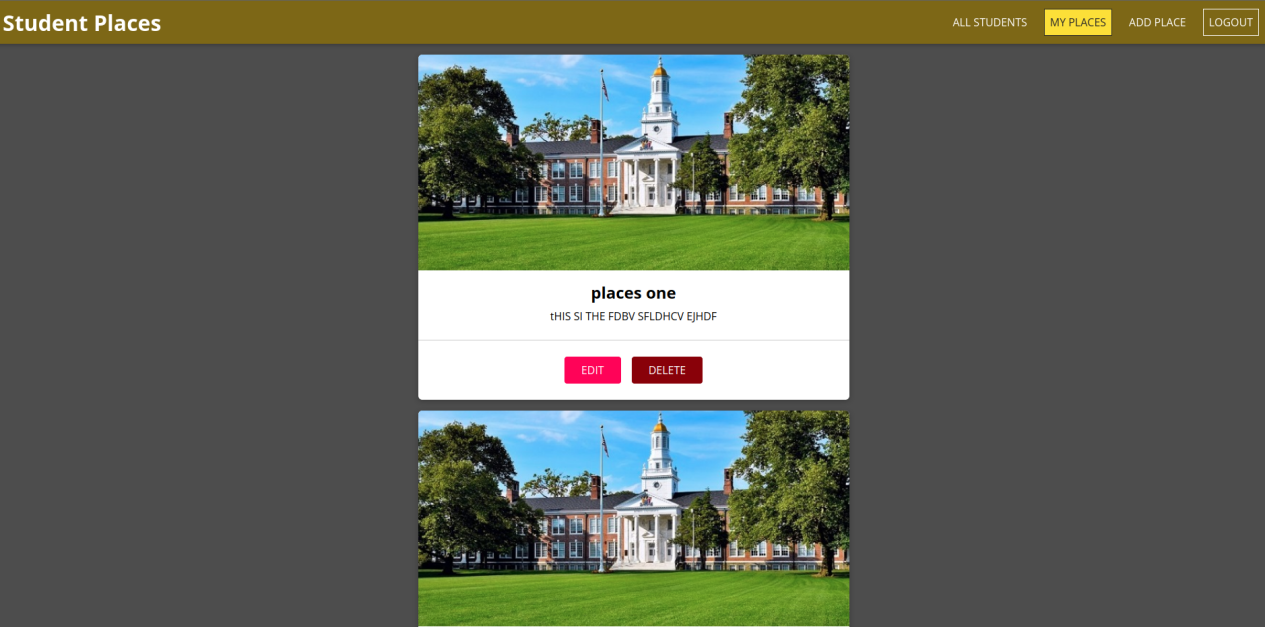


Screenshot Description:

The attached screenshot displays the documents in the "places" collection within MongoDB. Each document includes details such as the title, description, image URL, address, creator ID, and location coordinates. This visual representation helps in understanding the data structure and how it corresponds to the PlaceItem component in the application.

**Browser PlaceList Screenshot(Frontend/UI):**

To demonstrate the user interface and how the list of places is displayed in the application, a screenshot from the browser is attached. This screenshot shows the rendered list of places as seen by the end-user.



**Part Two: Exploring Fetch API and CORS Configuration in Node/Express**

Introduction:

The Fetch API and CORS (Cross-Origin Resource Sharing) are crucial aspects of modern web development, particularly when dealing with client-server communication in a distributed environment. This document explores how to utilize the Fetch API for making HTTP requests from a client-side application to a Node.js/Express server, and how to configure CORS on the server to handle these requests securely.

**Step 1: Fetch API**

Overview:

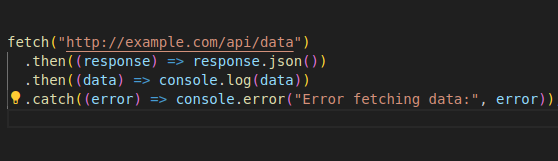
The Fetch API provides an interface for fetching resources (such as JSON data, images, files) across the network. It is built into modern browsers and provides a more powerful and flexible way to make HTTP requests compared to its predecessors (like XMLHttpRequest).

Using Fetch API in Client-Side Applications

Basic Fetch Request

To initiate a basic GET request using the Fetch API in a client-side application (e.g., React, Angular, Vue.js), you can use the following

syntax:



Making POST Requests

Fetching data with POST requests involves additional options, such as specifying headers and sending data:

Syntax of Post Request:



**Step 2: CORS (Cross-Origin Resource Sharing)**

CORS is a security feature implemented by web browsers to restrict web pages from making requests to domains other than the one that served the original web page. It is enforced by the browser and provides a way for servers to specify who can access their resources.

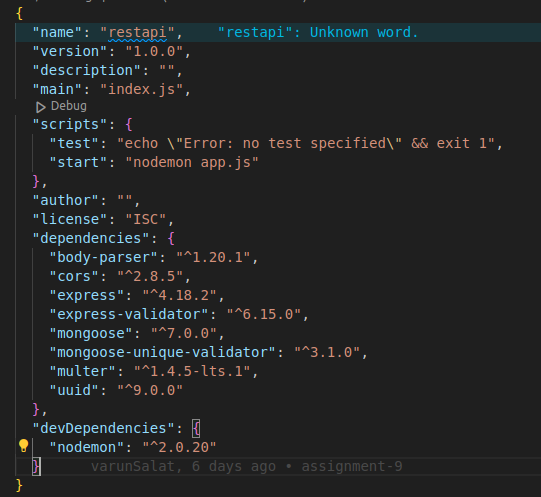
A: CORS Configuration in Node.js/Express

CORS Middleware Installation

In a Node.js/Express server, CORS handling is typically managed using middleware.

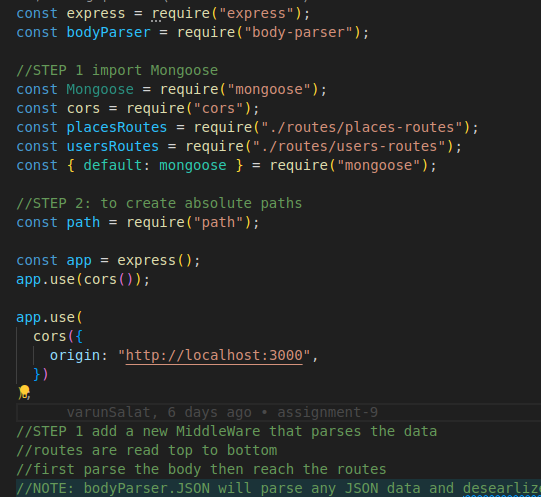
Install the cors package from npm: npm i cors

I will be seen in dependencies tree after installation:



B: Implementing CORS Middleware

Configure CORS middleware in Express application to allow or restrict cross-origin requests based as per requirements. Here’s how we can set up CORS middleware:



C: CORS Options:

origin: Specifies which domains are allowed to access the resources. You can use a string for a single origin or an array for multiple origins.

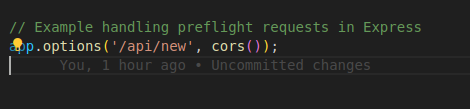
methods: Specifies which HTTP methods are allowed for cross-origin requests.

headers: Specifies which headers can be used during the actual request.

credentials: Indicates whether the request can include credentials like cookies or HTTP authentication information.

D: Handling Preflight Requests:

For certain types of requests, such as those with custom headers or methods other than GET or POST, the browser sends a preflight request (OPTIONS) to determine whether the actual request is safe to send. The server needs to respond to these preflight requests appropriately.



**Conclusion of Part Two:**

Understanding how to effectively use the Fetch API and configure CORS in Node.js/Express is essential for building modern web applications that interact securely and efficiently with server-side resources. By following these guidelines, we can ensure smooth client-server communication while adhering to security best practices.