MERN Stack & GeoJSON

Workshop

OVERVIEW

In this exercise, you will have the opportunity to build your first full stack application using the MERN stack (MongoDB, Express, React, Node.js)

GOALS

1. Creating a MERN stack CRUD application
2. Exploring MongoDB's geospatial queries
3. Using react-leaflet to create dynamic maps

SPECIFICATIONS

Create a real estate application where users can search for properties based on their desired location. Implement mongoose's geo location search to enable users to find properties within a specific area or proximity to certain landmarks.

Exercises

Setup

- Create a github repository and clone it
    - make sure it was initialized with a README.md file and a .gitignore file for node
- Initialize the package.json inside the root directory
- Create a cluster on mongoDB Atlas if you haven't got one running

Exercise 01

- Create a basic express server
    - use express.json() middleware
    - use the cors middleware
    - setup environment variables using dotenv
    - create the `routes`, `controllers`, and `models` directories
- Create a `db.js` file and setup a connection to the db and require it in index.js
- Create a react application and setup the routing
- Create a `models/user.js` file that contains the schema and model for the users collection
    - **Create** the **userSchema** with the following fields
        - **email** => String, Unique, Required
        - **name** => String, Required
        - **phoneNumber** => String, Unique, Required
    - **Create** and **export** the user model
- Create a `models/property.js` file that contains the schema and model for the users collection
    - **Create** the **propertySchema** with the following fields
        - **title**=> String, Unique, Required
        - **Description**=> String, Required
        - **price**=> Number, Required
        - **bedrooms**=> Number, Required, min: 1
        - **area=>** Number, Required
        - **image=>** String, Required
        - **images=>** Array of Strings
        - **owner=>** ObjectId, ref: UserModel (use the same name you gave the user model)

        - **availability=>** String, enum: ['vacant', 'rented', 'sold'], default: 'vacant'
        - **createdAt=>** Date, default: Date.now
    - **Create** and **export** the propertymodel

Exercise 02

- Create an endpoint that accepts a **POST** request on path **'/users'** to save a new user in the database (test it using insomnia)
    - creates **usersRouter** inside **`routes/users.js`** and use it in **`index.js`**
    - create the **createUser** function inside **`controllers/users.js`** and use it in the **`routes/users.js`** file
- Create an endpoint that accepts a **POST** request on path **'/properties** to save a new property in the database (test it using insomnia)
    - creates **usersRouter** inside **`routes/properties.js`** and use it in **`index.js`**
    - create the **createProperty** function inside **`controllers/properties.js`** and use it in the **`routes/properties.js`** file
- Create an endpoint that accepts a **GET** request on path **`/properties`** to retrieve all properties from the database, it should **populate** the **owner** filed as well
- create the **getProperties** function inside **`controllers/properties.js`** and use it in the **`routes/properties.js`** file
- Create an endpoint that accepts a **GET** request on path **`/properties/:id`** to retrieve the property depending on the id from the database, it should **populate** the **owner** filed as well
- create the **getProperty** function inside **`controllers/properties.js`** and use it in the **`routes/properties.js`** file

Exercise 03

- In the frontend create a component that displays the list of properties from the db when on the route **`/properties`**
    - Display the title, owner name, price, area, bedrooms, and the image for every property
- In the frontend create a component that displays the property details when on **`/properties/:id`**
    - Display all the fields + the contact information from the owner

Exercise 04 (Read Carefully)

- Modify **propertySchema** to add a new field called **location** that will be used to save geospatial coordinates and query based on radius
    - Read about geoJSON and create a **pointSchema** like in the example
    - Add the **pointSchema** as the type for the **location** field like shown in the example
    - Read the **Geospatial Indexes** section at the end and add the **index** property to the **location**
- Create the index in the users collection, this could be done using the atlas console or by adding `**ModelName.collection.createIndex({ location: '2dsphere' })**` before you export the model
    - **createIndex** will create an index on the selected model, notice that it is used on the model and not the schema
    - You can view the **indexes** that you have on the collection by opening mongodb atlas and navigating to the collection, click the **indexes** tab to view and verify that the index was added correctly, from there you can create and delete indexes
- Create an endpoint that accepts a **GET** request on path `**/properties/near-by**` to retrieve the properties from the db depending on the query parameter `**/properties/near-by?lng=52.5200&lat=13.4076&distance=1000**`
    - use $near inside of `**.find()**` to search for users based on the distance
    - the **lng** and **lat** query parameters representing the area you are trying to search from and the distance is the radius around that point, the distance value is in meters
    - use the **$maxDistance** to search for document within the max distance

Key takeaways from Exercise 04:

- Indexes in MongoDB are used to optimize query performance and improve overall database efficiency
- 2dsphere index is a geospatial index in MongoDB designed to support queries that involve geometric shapes on a sphere, such as points, lines, and polygons on the Earth's surface. The 2dsphere index is optimized for performing geospatial

queries using spherical geometry and can efficiently handle queries for proximity, containment, and intersection of geospatial objects

Exercise 05

- In the frontend create a component that displays the list of properties from the db based on the distance from the user when on the route `**/properties/near-by**`
    - get the user's geo location from the browser and send it in the request as query parameters
    - add a dropdown menu with different radius values to filter the properties (**NOT FRONTEND FILTRATION**), send a new request with the new radius to get the matching properties
- In the front create a components that displays the properties near by the user in a map when on the route `**/properties/map/near-by**`
    - use react-leaflet to render the map, use the users geo location to center the map at their location
    - add a dropdown menu with different radius values to filter the properties
    - get the properties based on the distance from the users location
    - add them as **<Marker/>** points in the map
    - provide a ` **<Popup></Popup>**` for each marker with the **title** of the property
    - when the popup clicked navigate the user to the property details page