MongoDB & Mongoose Exercise

Exercises

OVERVIEW

In this exercise, you will have the opportunity to practice using Mongoose, a flexible MongoDB object modeling (ODM) library for Node.js. You will focus on implementing CRUD (Create, Read, Update, Delete) operations and exploring the concept of referencing other documents within a MongoDB database.

GOALS

1. Introduction to Mongoose
2. Creating Mongoose Models and Schemas
3. Implementing CRUD Operations
4. Understanding Referencing documents in MongoDB

SPECIFICATIONS

In this exercise, you will create an Event Management System that allows users to manage and organize events. The system will provide features for creating events and managing them.

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
    - setup environment variables using dotenv` package
        - Add the mongodb connection string as an environment variable, make sure the connection string has the credentials of a user that has access to the cluster
        - Make sure the cluster allows connection from anywhere or at least from your own ip address
- Create a `db.js` file and setup a connection to the db
    - Check the documentation if you need help or reference the slides
- Require the db connection file in index.js
    - If the connection fails you make sure that the environment variable has a value, if you require db,js before dotenv/config then it will be undefined
- 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
        - **age** => Number, min: 18,
        - **phoneNumber** => String, Unique, Required
        - **isActive =>** Boolean, default: true
    - **Create** and **export** the user model

Key takeaways from Exercise 01:

- Please note that for the purpose of this exercise, you are working in a "development" environment where certain database restrictions may be relaxed. However, in a "production" environment, it is of utmost importance to implement additional restrictions for database users and allowed IP addresses. This is essential for maintaining a secure and controlled environment that ensures the safe access and management of data
- The schema and model in Mongoose are important for defining the structure of the data, enforcing validation rules, and providing an easy-to-use interface for interacting with the MongoDB database

Exercise 02

- Create an endpoint that accepts a **POST** request on path '**/users**` to save a new user in the db
    - HINT: Use the user model to create a new user, check the documentation or slides for more information
- Create an endpoint that accepts a **GET** request on path '**/users**` to retrieve all the **active** users from the db
    - HINT: read the documentation for more information about querying
    - OPTIONAL: use query parameters to provide the value for the active query parameter `/users?active=true` or `/users?active=false`
- Create an endpoint that accepts a **GET** request on path '**/users/:id**` to retrieve the user with the matching url parameter from the db
- Create an endpoint that accepts a **PUT** request on path '**/users/:id**` to update the user with the matching url parameter from the db
- Create an endpoint that accepts a **DELETE** request on path '**/users/:id**` to update the user with the matching url parameter from the db

Key takeaways from Exercise 02:

- Mongoose models provide an easy way to access and modify the data in the db, here are some examples of the things that can be done using the Mongoose
    - Create, Read (includes Querying and Filtering), Update, and Delete Documents
    - Data Population, which allows to automatically retrieve and populate referenced documents from other collections
    - Middleware functions that are known as `pre()` and `post()` hooks that can run before or after a certain trigger

Exercise 03

- Create a `models/event.js` file that contains the schema and model for the events collection
    - **Create** the **eventSchema** with the following fields
        - **name**=> String, Required
        - **description**=> String, Required
        - **location**=> String, Required,
    - Update the schema to add a new field named `organizer` that references a user user from the users collection
        - mongoose.Schema.Types.ObjectId is a mongoose type for document objectId
        - the ref key takes the name of the model name of the other collection
- **Create** and **export** the event mode

Key takeaways from Exercise 03:

- referencing other documents enhances data integrity, reduces redundancy, and provides a more flexible and scalable data model

Exercise 04

- Create an endpoint that accepts a **POST** request on path '**/events**` to save a new event in the db
    - make sure that the objectId belongs to a user
- Create an endpoint that accepts a **GET** request on path '**/events**` to retrieve all the events from the db
- Create an endpoint that accepts a **GET** request on path '**/event/:id**` to retrieve the event with the matching url parameter from the db
    - Update the endpoint to populate the organizer field so it brings the user document as well

Key takeaways from Exercise 04:

- The populate method is used to fetch the referenced document(s) from other collections and populate them in place of the specified field. It allows you to retrieve the complete referenced document(s) instead of just the ObjectId(s) or the partial information stored in the current document
- As mentioned above it is possible to reference multiple documents in one field, that can be accomplished by having an array of objectIds => `filedName: [{ type: Schema.Types.ObjectId, ref: 'ModelName'}]

  `

Exercise 05

- Update the event schema to have a new field named `attendees` that references some users who are attending the event
- Create an endpoint that accepts a **PATCH** request on path '**/events/:id/join**` to add a user to the attendees array for the event with the matching url parameter from the db
  - HINT: send the user id using the request body, you could use the $push to push a new value to the array or you could fetch the event then update it using `Array.push()`and then save it using `.save()`

Key takeaways from Exercise 05:

- **PATCH** is used for updating specific fields in a document, while **PUT** is typically used for updating the entire document, expecting all fields to be passed even if they have unchanged values
- There are multiple approaches to achieve the same outcome in programming, each with its own pros and cons. In the previous example, you can update a field directly using findByIdAndUpdate and $push, or you can fetch the document using findById and update it using .save().
- Exploring and understanding different ways of accomplishing tasks enhances your code reading skills and deepens your understanding of the topic