CMPS 356 Enterprise Application Development - Spring 2019
Lab 9 – Mongo DB

Objective

The objective of this lab is

- Understand Mongo Database and how to use it
- Use mongoose to create document schema and model
- Use mongoose to read/write MongoDB documents to implement CRUD operations
- Practice MongoDB aggregation queries

Overview

This Lab is based on Lab 78Banking App and Bookstore App. You are required to implement MongoDB repositories for both applications. DB repositories you will implement and deliver the same functionality as the File-based repositories provided in the base solution.

The tasks for this Lab are:

A. Implement and test the Banking App database schema and repository methods– **[1.2 hrs]**
B. Implement and test the Bookstore APP database schema and repository methods [**1.2 hrs**]

Project Setup

1. Download "Lab9-MongoDB" from the GitHub Repo and copy it to your repository.
2. Ensure that your **WebStorm** JavaScript language is set to **ECMAScript 6** and **Node.js Core** Libraries are enabled.
3. Make sure you have MongoDB installed. [https://www.mongodb.com/download-center/community]
4. Make sure MongoDB Compass is installed [https://www.mongodb.com/products/compass]
5. The project should be organized as follows:
    a. **public** folder which contains HTML pages, templates, CSS and client-side JavaScript
    b. **data** folder has JSON files to be used in this lab.
    c. **repositories** folder contains the controller classes.
    d. **models** folder contains the repository classes.
    e. **services** folder which contains all the services

PART A - Banking App

Open the **BankingApp** on Webstorm and follow the steps below.

I. Connecting to Database Using Mongoose

1. Open the terminal and start MongoDB server using **mongod**
2. Connecting to the Mongo Db Using mongoose
    - Install the mongoose package using the npm *[**npm** install **mongoos**]*
    - Open your terminal and type "mongo" . You should see the following

```
MongoDB shell version v3.4.1
connecting to: mongodb://127.0.0.1:27017
```

- Open your app.js and import the **mongoose** package
- Use mongoose to connect to the database link above (if the database does not exist then it will be auto created)

```
mongoose.connect('mongodb://localhost/BankDB', {useNewUrlParser: true , useNewUrlParser: true
useCreateIndex: true});
```

If connecting fails on your machine, try using `127.0.0.1` instead of `localhost`.

II. Creating the Database Schemas , Models and Entities

Before diving into the code of the application, it's always a good idea to map the relationships between entities and how to handle the data that must pass between them. A UML (Unified Modeling Language) diagram is a straightforward way to express relationships between entities in a way that can be referenced quickly as you type them out. This is useful for the person laying the groundwork for an application as well as anyone who wants to additional information in the database schema to it. An UML class diagram could appear as such:

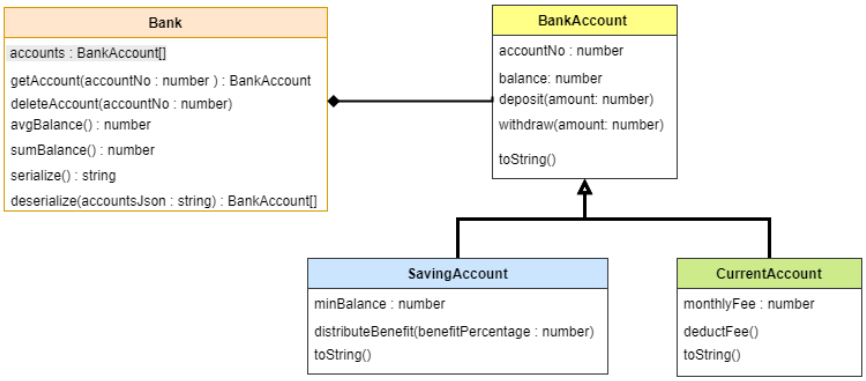

Figure 1 Banking App Class Diagram

1. Create a new folder and name it ***models***
2. Inside the ***models*** folder create two files and name them "**account.js**" and "**account-transaction.js**"
3. Inside **account.js** create **accountSchema** with the following properties
   - **accountNo** should be a string and it is required
   - **acctType** should be a string and its value could be either Saving or Current.
   - **balance** should be a number and it is required and with a custom validation error "Balance is a required property"
4. Add a virtual property **minimumBalance** that returns 1000 if the account type is Saving or null otherwise.
5. Create and export a Model named **Account** based on the **accountSchema**
6. Inside **account-transaction.js**. create **accountTransactionSchema** with the following properties
   - **transactionType** is a String and it is required. Its value could be either Debit or Credit
   - **amount** is a Number and it is required.
   - **accountNo** is a reference to the **Account** model.
7. Create and export a Model named **AccountTransaction** based on the **AccountTransactionSchema**
8. Open the **account-repository.js** and Import both the **Transaction** and **Account** Models

9. Implement all the repository methods using the **Transaction** and **Account** Models.
10. Add a route for the **sumBalance**.
11. Implement the **sumBalance** using aggregation function.
12. Test each method using **Mocha** or **Postman**.

PART B – Book Store App

In part B you should implement the Database Schema and the repository for the **BookStore App**. The DB repository you will implement will deliver the same functionality as the File-based repository provided in the base solution with minor modifications. NOTE : **You should test your implementation as you progress and document your testing .**

1. Open the **Book Store APP**
2. Change app.js to connect to "**BooksDB**" MongoDB database
3. Create three Models [Book, Author , Borrower]

    a. **Books Model** : This model should contain all the book related information such

    "title : "isbn : "pageCount":"publishedDate":"thumbnailUrl": "shortDescription":

    "longDescription": ""status":

    "authors": *[*list of author ids*]*, " borrower ": *borrower id*, "categories": []

    b. **Author Model**: This model should contain the following properties

    ["name" , "phoneNumber" , "street" , "email" , authoredBooks :[list of book ids]]

    c. **Brower Model**: This model should also will have the same fields as the author **["name" , "phoneNumber" , "street" , "email" , "borrowedBooks": [list of books ids]]**

    ➔To better understand the fields data-type refer the json files inside the data folder

    d. Make sure both the Book Model , Borrower Model and Author Model schema properties are Validated with custom validation.

    e. After you finish creating your models, open the **books-repository.js** and Import both models **Author and Books**.

    f. Implement **all the methods in books-repository.js using the Book and Author** models.

    Note: The borrower model is a new model, so you should add the necessary code to your repo, service and routes that should allow a person to borrow a book. When a borrow method is called you should

        i. Send the information of the **borrower** and the **book** they want to borrow

        ii. Add the **book ID** to the **borrower** model

iii. Add the borrower id to the book they borrowed

NOTE: All the query should be done on the Database. You should NOT do any filtering or aggregation on the client-side using JavaScript. You should use the database queries capabilities to implement all the aggregation, filtering needed for your solution.

Further details about MongoDB query operators is available at
https://docs.mongodb.org/manual/reference/operator/query/

You may use Compass or Robo 3T to interact with MongoDB database.

You need to test your implementation as you progress and document your testing. After you complete the lab, fill in the *Lab9-TestingDoc-Grading-Sheet.docx* and save it inside Lab9-MongoDB folder. Sync your repository to push your work to GitHub.