Working with Mongoose - I

Understanding Mongoose

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a structured way to interact with your MongoDB database, making it easier to work with data and perform database operations using JavaScript.

Importance of Mongoose

Mongoose holds a crucial role in MongoDB and Node.js development for several reasons:

- **Structured Data Modeling**: Mongoose introduces a structured way to define data models using schemas. This structured approach ensures that your data adheres to a specific format and maintains data consistency. Implementing validations in Mongoose schemas is crucial for maintaining data integrity and ensuring the overall health of your application.

- **Data Validation**: Defining validation rules for each field in your schema is vital for ensuring data quality and preventing invalid or malicious data from entering your database. This level of validation is a crucial security measure in real-world applications.

- **Middleware Support**: Middleware functions provided by Mongoose offer developers an invaluable tool to intercept and manipulate data before or after saving it to the database, enabling them to perform tasks such as hashing passwords, logging, and executing custom business logic.

- **Readable Query Building**: Mongoose simplifies constructing database queries using its query builder. This feature enhances code readability, making it easier to work with complex queries and improving maintainability.

- **Relationships and Population**: The ability to reference documents from other collections and retrieve them in a single query (known as "population") is essential for creating relationships between documents. This is vital when dealing with related data in your application.

- **Extensibility with Plugins**: Mongoose allows you to extend its functionality using pre-built plugins or creating custom ones. This promotes code reusability and allows developers to add features seamlessly.

Connecting using Mongoose

To use Mongoose, you need to install it using npm or yarn:

```
npm install mongoose
```

Once installed, you can connect to your MongoDB database using Mongoose:

```
Import mongoose from 'mongoose';

mongoose.connect('mongodb://localhost:27017/mydb', {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
```

NOTE: useNewUrlParser and useUnifiedTopology are optional parameters.
useNewUrlParser: When connecting to MongoDB using Mongoose, setting this option to true is recommended for newer MongoDB driver versions (3.1.0 and later) to ensure proper URL parsing.
useUnifiedTopology: It is also advisable to set it to true when connecting with newer MongoDB driver versions for improved server discovery and monitoring.

Creating Schemas

A Mongoose schema is a blueprint that defines the structure and constraints of documents within a MongoDB collection. It specifies the fields, their types, default values, and validation rules for data entering the database.

Critical Components of a Mongoose Schema

Field Definitions: In the schema, you define each field with a name and a data type (e.g., String, Number, Date, etc.). For example,

```
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
});
```

Validation Rules: You can enforce data validation rules using properties like required, min, max, enum, and custom validation functions. This ensures that data meets your application's requirements. Example:

```
const userSchema = new mongoose.Schema({
  name: {
    type: String,
    required: true,
```

```
  },
  age: {
    type: Number,
    min: 18,
  },
});
```

Default Values: You can specify default values for fields, and Mongoose will use them if you do not provide a value during document creation. For example:

```
const userSchema = new mongoose.Schema({
  role: {
    type: String,
    default: 'user',
  },
});
```

Complex Types: Mongoose allows you to define fields with complex types, such as arrays or nested objects. Example:

```
const userSchema = new mongoose.Schema({
  hobbies: [String],
  address: {
    street: String,
    city: String,
    zip: String,
  },
});
```

User operations

User operations are fundamental actions performed within applications to manage user data, enabling personalised experiences and secure access. These operations

include creating, retrieving, updating, and deleting user records.

Performing basic user operations

1. Creating a User (Create - C)

To create a new user document, define a Mongoose model based on a schema:

```
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
});
const User = mongoose.model('User', userSchema);
```

Create a new user by instantiating a User model object and then saving it to the database using async/await:

```
async function createUser() {
  const newUser = new User({
    name: 'John Doe',
    age: 30,
  });

  try {
    const savedUser = await newUser.save();
    console.log('User saved successfully:', savedUser);
  } catch (error) {
    console.error('Error saving user:', error);
  }
}

createUser();
```

2. Reading Users (Read - R)

Retrieve users from the database using async/await with the find() method. For example, to find all users:

```
async function findAllUsers() {
  try {
    const users = await User.find({});
    console.log('All users:', users);
  } catch (error) {
    console.error('Error finding users:', error);
  }
}

findAllUsers();
```

To find a specific user by a certain condition (e.g., by name), use a query object with async/await.

3. Updating a User (Update - U)

Update a user by retrieving the user you want to update, modifying its properties, and then saving it to the database using async/await. For example, updating a user's age:

```
async function updateAge() {
  try {
    const user = await User.findOne({ name: 'John Doe' });
    if (user) {
      user.age = 31; // Update age
      const updatedUser = await user.save();
      console.log('User updated successfully:', updatedUser);
    } else {
      console.log('User not found.');
    }
  } catch (error) {
    console.error('Error updating user:', error);
  }
```

```
}

updateAge();
```

4. Deleting a User (Delete - D)

Delete a user using async/await with the deleteOne() method. For example, to delete a user by name:

```
async function deleteUser() {
  try {
    const result = await User.deleteOne({ name: 'John Doe' });
    if (result.deletedCount > 0) {
      console.log('User deleted successfully.');
    } else {
      console.log('User not found.');
    }
  } catch (error) {
    console.error('Error deleting user:', error);
  }
}

deleteUser();
```

Summarising it

Let’s summarise what we have learned in this module:

- We introduced Mongoose as an ODM library, simplifying MongoDB interactions in Node.js applications.
- We explored the significance of Mongoose in MongoDB and Node.js development.
- We discussed the steps to install and connect Mongoose to a MongoDB database.
- Explored the concept of schemas in Mongoose, which define document structure, validation rules, default values, and support complex types.
- We introduced fundamental user operations—Create, Read, Update, Delete—and demonstrated how to perform them using Mongoose models and async/await

Some Additional Resources:

- **Validations in Mongoose**
- **Queries using Mongoose**