Exploration — Using Mongoose For MongoDB

Introduction

We have already used the `mongodb` npm package in our Node app to access data stored in MongoDB. In this exploration, we will study Mongoose, which is a JavaScript library that provides a syntactic layer between the data in a database's collection of documents and the objects in a Node app. Mongoose makes it easier for our app to interact with MongoDB. However, before studying how to use Mongoose, we will take a first look at the MVC design pattern, a pattern that is widely used in apps that have a graphical user-interface (GUI). The MVC pattern is used both for conceptually decomposing an app into different layers, as well as, for organizing the code for easier maintainance.

The MVC Pattern

MVC stands for Model-View-Controller. This design pattern was first introduced in the Smalltalk project back in late 1970's. The basic idea is that we decompose the functionality of an app with a GUI into 3 layers, model, view, and controller.

Model

The model of an app corresponds to the layer that manages the application's data. In apps developed using languages with support for object-orientation, the model will include:

- Classes in the app that represent the data in the database.
- Mapping from the MongoDB documents to JavaScript objects.
- Mapping from the JavaScript objects to MongoDB documents.
- Code that executes CRUD operations on the DBMS.

Note that sometimes the term **models** is used to refer to just the classes that represent the data in the database.

View

The view is the layer of the app that displays data from the app. In a web app, the view is ultimately rendered in HTML and CSS. In a desktop app, the view will be rendered using a UI toolkit for the platform on which the app is running. The view does not directly interact with the model. Instead, it will send information about a user's interaction with the view to the controller.

Note that sometimes the term **views** is used to refer to the rendered displays of data. For example, different pages in a web app may be referred to as views of the web app.

Controller

The controller is the layer of the app that connects the view to the model. The controller handles requests from the view layer. It determines how to process the request. It decides how to involve the view and model layer in processing the request, including how the view and the model might be updated due to the request.

In an Express app, the controller layer is implemented by the route handlers. The route handlers:

- Receive requests
- Call functions to perform CRUD operations on the model layer
- Send back responses that update the view

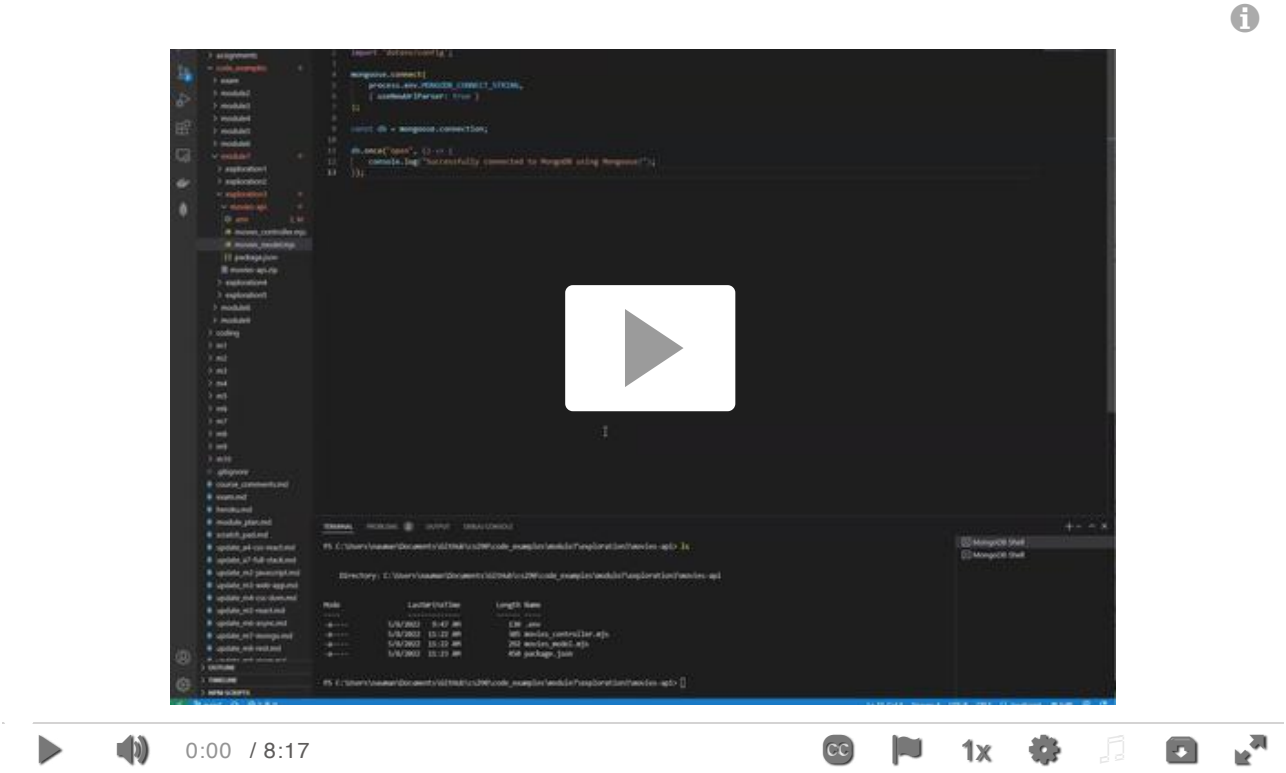
Note that sometimes the term **controllers** is used to refer to the various route handlers in an Express app.

Using the MVC Pattern

We suggest that you use the MVC pattern as a high-level guideline to think about how to decompose your app into maintainable layers, so that changes can be carried out in one layer without impacting the other layers. If you read about the MVC pattern, you will find many variations

of the pattern. Do not get bogged down by trying to strictly adhere to some particular variation of the pattern.

Using Mongoose to Connect to MongoDB



Mongoose is used in the model layer of a web app. It is an object-document mapper (ODM) that maps between classes and objects in our JavaScript code and the documents stored in MongoDB. To use Mongoose, we will use the `mongoose` npm package. To add some structure to our code, instead of putting all the code in one file, we will create separate files for the model layer and the controller layer.

Download the upZip this **movies-api.zip (https://canvas.oregonstate.edu/courses/1879154/files/94479023?wrap=1)** (https://canvas.oregonstate.edu/courses/1879154/files/94479023/download?download_frd=1) archive to use with this Exploration and the next. In the terminal, run the commands `npm install` and `npm start`. This zip file contains the following 5 files:.

- `movies-model.mjs`
  - This file contains the model layer.
  - It uses the npm package `mongoose` to connect to MongoDB.
  - This file doesn't import the `express` package because the model layer should be independent of the controller and shouldn't use the Express API.
  - We will add functions in this file to implement the CRUD operations. We will export these functions to make them available for use outside this file.
- `movies-controller.mjs`

    - This file contains the controller layer.
    - It will include our Express code to implement the route handlers.
    - This file doesn't use any mongoose APIs and doesn't import the `mongoose` package because all the direct interaction with the database must be done by the model layer.
    - It imports `movies_model.mjs`. The route handlers we will implement will call the relevant functions in `movies_model.mjs`.
- `.env`
    - The environment file with 2 variables.
        - PORT=3000
        - MONGODB_CONNECT_STRING='Connect string your MongoDB server goes here'
    - Don't update the value of PORT. But you do need to updated the value of `MONGODB_CONNECT_STRING` to the connect string for your MongoDB server
- `package.json`
    - The new package dependency in this file is `mongoose`. We have used all the other packages before.
- `test-requests.txt`
    - Provides URLs for creating, retrieving, updating, and deleting documents from the collection.

Connecting to MongoDB

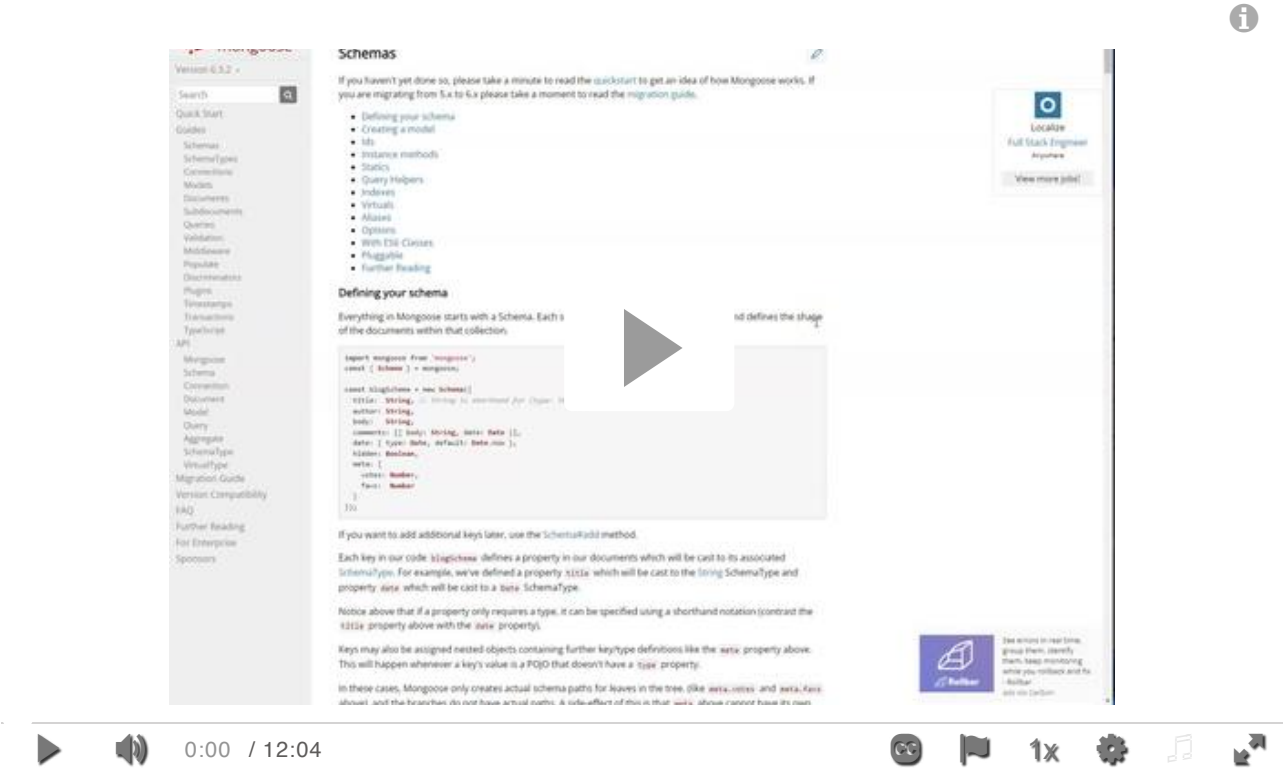
The following code imports the `mongoose` package, connects to the MongoDB server and database specified in the variable `MONGODB_CONNECT_STRING` in the file `.env`.

```
import mongoose from 'mongoose';
import 'doteven/config';

mongoose.connect(
    process.env.MONGODB_CONNECT_STRING,
    { useNewUrlParser: true }
);

const db = mongoose.connection;

db.once("open", () => {
    console.log("Successfully connected to MongoDB using Mongoose!");
});
```

Schemas and Models

> **Note:** For a quick refresher on Object-Oriented Programming in JavaScript, see the **relevant exploration in Module 2 (https://canvas.oregonstate.edu/courses/1879154/pages/exploration-object-oriented-programming)** . Pay particular attention to the sections **Defining Classes** and **Static Methods and Fields**.

In Mongoose, schemas and models are the main concepts for mapping between documents in MongoDB and objects in our JavaScript code.



Schema

In Mongoose, a schema represents the properties of a collection in MongoDB. We define a schema by calling the method `Schema` on the `mongoose` object.

- For each property, we specify the data type.
- **Data types** (https://mongoosejs.com/docs/schematypes.html) (formally called Schema Types), include `String`, `Number`, `Boolean`, `Date`, `Array`, and `Map` (key-value pairs), among others types.
- We can specify certain constraints on individual properties. These are called **schema type options** (https://mongoosejs.com/docs/schematypes.html#schematype-options). For example:
  - We can declare that specifying the value of a property is optional.
  - Or we can declare a property to be `required`. This will mean that the every document must include a value for that property. Otherwise, Mongoose will not save the document to MongoDB.

Example: Schema Definition

In the following example, we define a schema for a MongoDB document to represent movies. The schema has three properties, `title`, `year` and `language`. All three of these properties are required.

```
const movieSchema = mongoose.Schema({
        title: { type: String, required: true },
        year: { type: Number, required: true },
        language: { type: String, required: true }
});
```

Mongoose Model

In Mongoose, the *model* is a JavaScript class which represents documents of a particular collection.

- Mongoose generates this JavaScript class for us.
- To do this, we need to call the method `model` on the `mongoose` object passing it two parameters
  - The name of the JavaScript class that Mongoose will generate.
  - The schema which Mongoose will use to generate this class.
- Once the model class is generated, we can add more methods to this class if necessary.
- We can use the model class to create documents by using JavaScript's built-in support for object-oriented programming, i.e., we can create documents by calling the class constructor with the `new` keyword.

Example: Creating Documents Using the Model Class

In the following code:

- When we call the function `mongoose.model()`, Mongoose generates a class named `Movie`.
  - This class corresponds to the `movieSchema` we defined earlier.
  - Objects of the class `Movie` have the properties `title`, `year` and `language` corresponding to the properties of the schema `movieSchema`.
- We then create an object of this class by calling the constructor with the keyword `new`

```
const Movie = mongoose.model("Movie", movieSchema);
```

We could then create a new movie this way:

```
const movie = new Movie({ title: "Fargo", year: 1996, language: "English" });
```

However, in the next Exploration, we'll review a detailed way using another method for the Model as well as tying it to the Controller.

Summary

In this exploration, we introduced the MVC pattern which gives us concepts to decompose the functionality of apps with a graphical user interface (GUI). We then introduced Mongoose, a popular tool for implementing the model layer of apps using MongoDB. In the next exploration, we will continue our study of Mongoose to learn how to use it to implement CRUD operations.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Here is a link for the **official Mongoose docs (https://mongoosejs.com/docs/guide.html) (https://mongoosejs.com/docs/)** . Unfortunately, neither the structure nor the writing style of these docs is very clear.
- The page on **Mongoose npm package (https://www.npmjs.com/package/mongoose)** gives a good overview of the functionality of this package.