Week 9 Practical: Mongoose

Just a document to go through and explore a bit of MongoDB integration

**Step 1: Setup**

* Open VS Code:
  + Make a server folder
  + Make an express server JavaScript file
  + Install: express, mongoose, express-validator (open an integrated terminal in the server folder)

Your repository should look like this:

A screenshot of a computer

Description automatically generated

Files we will make today (At the end of the lesson):

/server

**index.js**

/models

**books.js**

/routes

**book\_router.js**

/views

**add\_book.html**

Here is how I think of things:

1. When designing a product, investigate the user experience, what do they see, what pages will we need to have, and how are they going to interact with our application. You will build some understanding of what you need from the server from there.
2. What do you need from the server for the client?
3. How are we using the Database, if we are using one, and what are we storing?

When implementing what you’ve brainstormed, work in the opposite direction. What do you need to store on the DB, how does the server interact with both the DB and the client, create routes and your middleware, and finally move onto the client side (front end), as that is the most subjective.

Thought Process: client -> server -> db

Implementation: db -> server -> client

For today’s class we are making a bookstore implementation, we will create book entries into our DB, read data from the DB and delete entries. If we have time, I will have you work on your own to work on Updating entries.

What do we want to store:

* Book:
  + Title
  + Author
  + Pages
  + Genres – array
  + Rating

So, from the front end, we want to make a form that contains all this information and on the sever we want to store the content, read the existing content, and delete the content.

* Get all books
* Add a book
* Delete a book
* Update a book

(In the future we will implement authentication for users and authorizations to limit who can edit the books)

Since we are making a Server App, we will also need to create views for the forms we want to fill in. For now, I’ll write them in html, but I encourage you to investigate front-end technologies like react, vue, angular, htmx etc. By moving onto a frontend framework, the server’s role will need to be adjusted to sending information as an API

**Part 2: Making our server**

By now, we have made simple express servers like this:

require("dotenv").config();

const express = require("express");

const app = express();

const PORT = process.env.PORT || 8000;

app.use(express.urlencoded({ extended: true }));

app.use(express.json());

app.use((req, res) => {

  res.status(404).send("Route not found");

});

app.listen(PORT, () => console.log(`Server started on port ${PORT}`));

From here, we will try getting MongoDB connected onto our server. We can do this with a package called mongoose. It allows us to use MongoDB and its commands when interacting between the server and the Database. (Its an ODM – Object Data Modeling/Mapping Library)

More information: [Introduction to Mongoose for MongoDB (freecodecamp.org)](https://www.freecodecamp.org/news/introduction-to-mongoose-for-mongodb-d2a7aa593c57/)

FIRST: install mongoose: in the terminal: npm install mongoose

const mongoose = require("mongoose");

mongoose.connect();// ADD YOUR CONNECTION STRING HERE, FROM LAST WEEK

let db = mongoose.connection;

// Check connection

db.once("open", ()=> {

  console.log("Connected to MongoDB");

});

// Check for DB errors

db.on("error", (err)=> {

  console.log("DB Error:" + err);

});

Try to run the code at this point, if MongoDB connected, you should be all good to move to the next step

Possible errors:

Your IP is not recognized on Atlas, you need to go into network settings on Atlas and allow a new I:P address.

**Part 3: Creating a Model file**

A model file is a key component of the application’s data layer. The model file defines the structure of data that will be stored in the database and provides an interface for interacting with the data. A Model file helps with keeping data consistent and provide possible steps for validation.

Last week I had you build a bookstore database. We will now create a model for it so we can submit data for it through the DB.

Create a new file: book.js, store it into the model’s folder

Inside the file include this code.

* Import the mongoose package
* Define an object that contains the fields we want to store into our database
* Export the new schema -> we declare which collection to read from

let mongoose = require("mongoose");

let bookSchema = mongoose.Schema(

  {

    title: {

      type: String,

      required: true,

    },

    author: {

      type: String,

      required: true,

    },

    pages: {

      type: Number,

      required: true,

    },

    genres: {

      type: [String],

      required: true,

    },

    ratings: {

      type: Number,

      required: true,

    },

  },

  { collection: 'books'}

);

let Book = (module.exports = mongoose.model("books", bookSchema));

Now we have made the schema, we need to also add the schema to our server. Back in the index file, add this line (you can place it somewhere at the top, where you have declared variables):

const book = require("./models/book")

Next let’s try to grab information from our database. In our case, since we have the bookstore database, we should call to the books collection (table for the SQL people)

app.get("/", (req, res)=> {

  book

    .find({})

    .then((books) => {

      res.json(books);

    })

    .catch((error) => {

      console.error("Error:", error);

      res.status(500).json({ error: "Internal Server Error" });

    });

});

Upon receiving a home request, the server will be configured to send the book data from the bookstore. The book object is linked to our "books" collection, enabling us to execute commands on this collection. Using Mongoose, we can run these MongoDB commands effectively.

We did the find command last week to grab all the entries the collection, and we can do that here. Note, since MongoDB has gone into version 7.0, they have made changes to how you can grab this information, so some of the code written in here may break. They do not accept await async commands anymore, but they will accept promises, so up above is the promise command.

|  |
| --- |
| This command essentially says: Run the find command on the book collection, with the eventuality that I will either receive a response of either accomplished with the data, which I need to breakdown OR a reject, and I must work through the error log. |

Your current file should look something like this:

A screen shot of a computer

Description automatically generated

Here is the current file we have made for our server:

|  |
| --- |
| require("dotenv").config();  const express = require("express");  const mongoose = require("mongoose");  const PORT = process.env.PORT || 8000;  const book = require("./models/book");  const app = express();  app.use(express.urlencoded({ extended: true }));  app.use(express.json());  mongoose.connect(process.env.DB\_HOST);  let db = mongoose.connection;  db.once("open", () => {    console.log("Connected to MongoDB");  });  db.on("error", (err) => {    console.log("DB Error:" + err);  });  app.get("/", (req, res) => {    book      .find({})      .then((books) => {        res.json(books);      })      .catch((error) => {        console.error("Error:", error);        res.status(500).json({ error: "Internal Server Error" });      });  });  app.listen(PORT, () => console.log(`Server started on [http://127.0.0.1:${PORT}`)](http://127.0.0.1:$%7bPORT%7d%60))); |

**Part 4: Adding CRUD operations to our application:**

Just like last class, we should add some other operations for inserting and deleting information from the server.

This might take some work, but the idea would be:

1. Create an html form to submit data
2. Once we receive the data, parse the information, **validate the items**, and store it into a data object for us to submit the data into the DB
3. Handle the data and send it to MongoDB

Thought Process:

* Let’s create a route called “/add”
* Create a route to send over an html page (GET)
* Create a post method to handle the form data (POST) and once the request has been received:
  + Validate each item being submitted
  + Respond back with any errors if they exist
  + Bundle the data into a Book object
  + Send over the data to be saved into the database

Html page to handle the Form – you can make one online or use this (save in views folder, add.html)

<!DOCTYPE html>

<html lang="en">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-scale=1.0">

  <title>Add Book</title>

</head>

<body>

  <h1>Add a New Book</h1>

  <form action="/add" method="POST">

    <label for="title">Title:</label>

    <input type="text" id="title" name="title" required><br><br>

    <label for="author">Author:</label>

    <input type="text" id="author" name="author" required><br><br>

    <label for="pages">Pages:</label>

    <input type="number" id="pages" name="pages" required><br><br>

    <label for="rating">Rating:</label>

    <input type="number" id="rating" name="rating" min="0" max="5" required><br><br>

    <label for="genres">Genres:</label>

    <select id="genres" name="genres" required>

      <option value="Fiction">Fiction</option>

      <option value="Non-fiction">Non-fiction</option>

      <option value="Fantasy">Fantasy</option>

      <option value="Science Fiction">Science Fiction</option>

    </select><br><br>

    <button type="submit">Add Book</button>

  </form>

</body>

</html>

Server handling the request

// add route

const { body, validationResult } = require('express-validator');

app

  .route("/add")

  .get((req, res) => {

    res.sendFile(path.join(\_\_dirname, 'views/add.html'));

  })

  .post(

    [

      body("title", "Title is required").notEmpty(),

      body("author", "Author is required").notEmpty(),

      body("pages", "Pages is required").notEmpty(),

      body("rating", "Rating is required").notEmpty(),

      body("genres", "Genre is required").notEmpty(),

    ],

    (req, res) => {

      const errors = validationResult(req);

      if (!errors.isEmpty()) {

        return res.status(400).json({ errors: errors.array() });

      }

      let book = new Book();

      book.title = req.body.title;

      book.author = req.body.author;

      book.pages = req.body.pages;

      book.genres = req.body.genres;

      book.rating = req.body.rating;

      book

        .save()

        .then(() => {

          res.json({ message: "Successfully Added" });

        })

        .catch((err) => {

          console.error(err);

          res.status(500).json({ error: "Internal Server Error" });

        });

    }

  );

Lets refactor the code a bit, since we just added 40 lines to the index file:

Lets move the validator code out of the request and into a function, then move it into its own file:

Part 1:

const validateBook = [

  body("title", "Title is required").notEmpty(),

  body("author", "Author is required").notEmpty(),

  body("pages", "Pages is required").notEmpty(),

  body("rating", "Rating is required").notEmpty(),

  body("genres", "Genre is required").notEmpty(),

  (req, res, next) => {

    const errors = validationResult(req);

    if (!errors.isEmpty()) {

      return res.status(400).json({ errors: errors.array() });

    }

    next();

  },

];

New Post request (cleaner):

  .post(validateBook, (req, res) => {

    let book = new Book();

    book.title = req.body.title;

    book.author = req.body.author;

    book.pages = req.body.pages;

    book.genres = req.body.genres;

    book.rating = req.body.rating;

    book

      .save()

      .then(() => {

        res.json({ message: "Successfully Added" });

      })

      .catch((err) => {

        console.error(err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

Part 2:

Move validator into a new file, make it modular. This is here just to showcase you can do it, not that you have to, since the validator for books isn’t something we do aside from this file, it will be pointless in the future, for other than cleaning the file.

New file in middleware folder:

const { body, validationResult } = require("express-validator");

const validateBook = [

  body("title", "Title is required").notEmpty(),

  body("author", "Author is required").notEmpty(),

  body("pages", "Pages is required").notEmpty(),

  body("rating", "Rating is required").notEmpty(),

  body("genres", "Genre is required").notEmpty(),

  (req, res, next) => {

    const errors = validationResult(req);

    if (!errors.isEmpty()) {

      return res.status(400).json({ errors: errors.array() });

    }

    next();

  },

];

module.exports = validateBook;

add this into server, remove validator code form server.

const validateBook = require('./middleware/addbook\_validator')

Lets try finding single book items, using params

We need to find a book, by its id, so we can use the parameter of “:id”. What else should we be able to do with a single book? View the book, edit the book and delete the book?

What do we do:

When we receive the id, we search for the book, the document that contains that id. Like delete, we can use the deleteOne operation to delete the entry. In the future, you might use something other than the id to find a book, or search through a DB. (Searching for multiple titles instead – “Harry Potter” bringing up the book series)

// id search

app.route("/book/:id").get((req, res) => {

  Book.findById(req.params.id)

    .then(book => {

      if (!book) {

        return res.status(404).json({ error: 'Book not found' });

      }

      res.json({ book: book, genres: base\_genres });

    })

    .catch(err => {

      console.error('Error fetching book by id:', err);

      res.status(500).json({ error: 'Internal Server Error' });

    });

}).delete((req, res) => {

  const query = { \_id: req.params.id };

  Book.deleteOne(query)

    .then(result => {

      if (result.deletedCount > 0) {

        res.json({ message: 'Successfully Deleted' });

      } else {

        res.status(404).json({ error: 'Book not found' });

      }

    })

    .catch(err => {

      console.error('Error deleting book by id:', err);

      res.status(500).json({ error: 'Internal Server Error' });

    });

});

What about modification? Editing the page, well, you may notice when we want to modify our user information, they place some interactable item for us BEFORE we can modify a document. (Modifying your user profile requires you to press an edit button for example). For us, we will call to a separate route and do the operation, a limiting factor with how this application works.

From this point onward, here is the code from what we have worked on so far, with some modifications (index.js).

require("dotenv").config();

const express = require("express");

const mongoose = require("mongoose");

const PORT = process.env.PORT || 8000;

const Book = require("./models/book");

const app = express();

const path = require("path");

app.use(express.urlencoded({ extended: true }));

app.use(express.json());

mongoose.connect(process.env.DB\_HOST);

let db = mongoose.connection;

db.once("open", () => {

  console.log("Connected to MongoDB");

});

db.on("error", (err) => {

  console.log("DB Error:" + err);

});

app.get("/api/books", (req, res) => {

  Book.find({})

    .then((books) => {

      res.json(books);

    })

    .catch((error) => {

      console.error("Error:", error);

      res.status(500).json({ error: "Internal Server Error" });

    });

});

app.get('/books', (req, res) => {

  res.sendFile(path.join(\_\_dirname, 'views', 'books.html'));

});

// add route

const validateBook = require("./middleware/addbook\_validator");

app

  .route("/book/add")

  .get((req, res) => {

    res.sendFile(path.join(\_\_dirname, "views/add.html"));

  })

  .post(validateBook, (req, res) => {

    let book = new Book();

    book.title = req.body.title;

    book.author = req.body.author;

    book.pages = req.body.pages;

    book.genres = req.body.genres;

    book.rating = req.body.rating;

    book

      .save()

      .then(() => {

        res.json({ message: "Successfully Added" });

      })

      .catch((err) => {

        console.error(err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

// id search

app

  .route("/api/book/:id")

  .get((req, res) => {

    Book.findById(req.params.id)

      .then((book) => {

        if (!book) {

          return res.status(404).json({ error: "Book not found" });

        }

        res.json({ book: book });

      })

      .catch((err) => {

        console.error("Error fetching book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  })

app

  .route("/book/:id")

  .get((req, res) => {

    res.sendFile(path.join(\_\_dirname, "views", "book.html"))

  })

  .delete((req, res) => {

    const query = { \_id: req.params.id };

    Book.deleteOne(query)

      .then((result) => {

        if (result.deletedCount > 0) {

          res.json({ success: true, message: "Successfully Deleted" });

        } else {

          res.status(404).json({ error: "Book not found" });

        }

      })

      .catch((err) => {

        console.error("Error deleting book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

// edit book route

app.get('/book/edit/:id', (req, res)=>{

  res.sendFile(path.join(\_\_dirname, "views", "edit-book.html"))

})

app

  .route("/api/book/edit/:id")

  .get((req, res) => {

    Book.findById(req.params.id)

      .then((book) => {

        if (!book) {

          return res.status(404).json({ error: "Book not found" });

        }

        res.json({ book: book });

      })

      .catch((err) => {

        console.error("Error fetching book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  })

  .post((req, res) => {

    let updatedBook = {

      title: req.body.title,

      author: req.body.author,

      pages: req.body.pages,

      genres: req.body.genres,

      rating: req.body.rating,

    };

    const query = { \_id: req.params.id };

    Book.updateOne(query, updatedBook)

      .then(() => {

        res.json({ message: "Successfully Updated" });

      })

      .catch((err) => {

        console.error("Error updating book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

app.listen(PORT, () =>

  console.log(`Server started on http://127.0.0.1:${PORT}`)

);

Part 5:

Reorganization/refactoring your code. Please go to blackboard/this GitHub link as I have posted a functional version of what we have worked on so far.

<https://github.com/HarmanSMann/WF1-TEMP-W9-Mongoose>

Making a Router Folder

By now, you notice that I have modified the code for each of the book routes to include “/book”. The idea here is organizing the existing routes into a separate file to keep the content focused and keep out index file as clean as possible.

Here are our routes in the index file:

* /books
* /api/books
* /book/add
* /api/book/:id
* /book/:id
* /book/edit/:id
* /api/book/edit/:id

Lets reformat the routes so they all go through the /book route

* /book
  + /
  + /api/books
  + /add
  + /:id
  + /edit/:id
  + /api/book/edit/:id

To get this started, there is another feature within express that would allow you to make a ‘router’ file. By specifying a new route, we can handle all incoming requests to a subroute -> “/book” and anything after that can be pushed onto the router file.

We will make a folder: **routes**, and a file called: **book\_router** -> what are we moving into this file?

We are handling CRUD operations; we are working on express validation and we need the model file to process requests. So those are the first things we will add

Here is our new base

const express = require("express");

const router = express.Router();

const path = require("path");

const Book = require("../models/book");

const { body, validationResult } = require("express-validator");

Copy all the routes from index, (the ones we listed above), and move them into here. You will need to refactor some of the items once it is in there. Instead of app, we call to router and pass router back to the index file.

Here is a completed version:

const express = require("express");

const router = express.Router();

const path = require("path");

const Book = require("../models/book");

const validateBook = require("../middleware/addbook\_validator");

router.get("/", (req, res) => {

  res.sendFile(path.join(\_\_dirname, "../views", "books.html"));

});

router.get("/api/books", (req, res) => {

  Book.find({})

    .then((books) => {

      res.json(books);

    })

    .catch((error) => {

      console.error("Error:", error);

      res.status(500).json({ error: "Internal Server Error" });

    });

});

// add route

router

  .route("/add")

  .get((req, res) => {

    res.sendFile(path.join(\_\_dirname, "../views/add.html"));

  })

  .post(validateBook, (req, res) => {

    let book = new Book();

    book.title = req.body.title;

    book.author = req.body.author;

    book.pages = req.body.pages;

    book.genres = req.body.genres;

    book.rating = req.body.rating;

    book

      .save()

      .then(() => {

        res.json({ message: "Successfully Added" });

      })

      .catch((err) => {

        console.error(err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

// id search

router.route("/api/book/:id").get((req, res) => {

  Book.findById(req.params.id)

    .then((book) => {

      if (!book) {

        return res.status(404).json({ error: "Book not found" });

      }

      res.json({ book: book });

    })

    .catch((err) => {

      console.error("Error fetching book by id:", err);

      res.status(500).json({ error: "Internal Server Error" });

    });

});

router

  .route("/book/:id")

  .get((req, res) => {

    res.sendFile(path.join(\_\_dirname, "../views", "book.html"));

  })

  .delete((req, res) => {

    const query = { \_id: req.params.id };

    Book.deleteOne(query)

      .then((result) => {

        if (result.deletedCount > 0) {

          res.json({ success: true, message: "Successfully Deleted" });

        } else {

          res.status(404).json({ error: "Book not found" });

        }

      })

      .catch((err) => {

        console.error("Error deleting book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

// edit book route

router.get("/edit/:id", (req, res) => {

  res.sendFile(path.join(\_\_dirname, "../views", "edit-book.html"));

});

router

  .route("/api/book/edit/:id")

  .get((req, res) => {

    Book.findById(req.params.id)

      .then((book) => {

        if (!book) {

          return res.status(404).json({ error: "Book not found" });

        }

        res.json({ book: book });

      })

      .catch((err) => {

        console.error("Error fetching book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  })

  .post((req, res) => {

    let updatedBook = {

      title: req.body.title,

      author: req.body.author,

      pages: req.body.pages,

      genres: req.body.genres,

      rating: req.body.rating,

    };

    const query = { \_id: req.params.id };

    Book.updateOne(query, updatedBook)

      .then(() => {

        res.json({ message: "Successfully Updated" });

      })

      .catch((err) => {

        console.error("Error updating book by id:", err);

        res.status(500).json({ error: "Internal Server Error" });

      });

  });

module.exports = router;

Inside the index.js file we need to make these modifications:

Import the new router:

const book\_router = require("./router/book\_router");

setup a new route to be used whenever we want access to the bookstore

app.use("/bookstore", book\_router)

This essentially makes it so that whenever we hit the routes, we need to go past like this:

localhost:8000/bookstore/ --- example: /add route for the bookstore

localhost:8000/bookstore/add

Aside from the other modifications, more towards code that hasn’t been refactored yet

Since we changed the routes, we must change the API calls from the html pages. Other than that, that’s the entirety of class today.

That is everything today:

We have accomplished:

* Connecting to MongoDB with Node/javascript
* Reading the collections
* Adding into our collection
* Editing into our collection
* Deleting into our collection

Next week:

A user system