LAB 14b

Working With Databases (Node)

What You Will Learn

- How to read data from SQLite in Node
- How to use and query data with MongoDB
- How to read Mongo data in Node using Mongoose
- How to use MongoDB Realms

Note

This chapter's content has been split into two labs: Lab14a and Lab14b.

Approximate Time

The exercises in this lab should take approximately 90 minutes to complete.

Fundamentals of Web Development, 3rd Ed

Randy Connolly and Ricardo Hoar

Textbook by Pearson
http://www.funwebdev.com

Date Last Revised: March 14, 2023

Preparing Directories

1 The starting `lab14b` folder has been provided for you (within the zip folder downloaded from Gumroad).

Working with SQLite

In this lab, you will be making use of different data sources in Node. To begin, you will use SQLite, a small relational database management system. Unlike more familiar database systems such as MySQL or Oracle, SQLite is a database engine built into and used by mobile apps, desktop applications, and, of course, Node on the server. SQLite is widely used (but hidden from the user) within applications such as cell phones, TVs, browsers, PCs, and Macs. According to the `https://www.sqlite.org` website, there are likely billions of SQLite databases in active use, making it by far the most used database engine on the planet.

A SQLite database is contained within a very small compressed single file. The drawback with the single file approach is that it can't handle numerous multiple simultaneous INSERT/UPDATE queries. But for small Node web apps (or ones that are mostly read-only with a low number of write requests), SQLite is fantastic since no other software is required on the server. With Node, you will simply add the sqlite3 package to your application.

Exercise 14b.1 — Setting Up SQLite

1 You do not need to install anything on your development machine or your production server to use SQLite. Nonetheless, you may find it convenient to install some of the SQLite tools on your development machine (though, again, there are not necessary) if you want to test some queries, modify the database, etc. There is a command-line shell program and an excellent GUI program named SQLite Studio.

2 If you do install any of these tools, you can examine the provided SQLite databases (`art.db`) in the `public` folder.

Exercise 14b.2 — Setting Up SQLite-Based Node App

1 In the terminal, make sure the current folder is where you want to create the node application. If it is, then type the following command:

```
npm init
```

It doesn't really matter how you answer these questions.

2 Enter the following commands:

```
npm install express
npm install sqlite3
```

Exercise 14b.3 — Using SQLite in Node

1 Create a new file named `db-tester.js`.

2 Add the following code:

```
const path = require("path");
// the verbose is optional but gives better error messages
const sqlite3 = require("sqlite3").verbose();
```

3 Add the following code:

```
// open the database
const DB_PATH = path.join(__dirname, "data/art.db");
const db = new sqlite3.Database(DB_PATH);
```

If you copied the starting code, you should have a folder named data with the sqlite database file in it.

4 Add the following code:

```
let sql = `SELECT GenreID,GenreName,EraID,Description,Link
            FROM Genres;`;
// retrieve all the data into memory
db.all(sql, [], (err, rows) => {
    if (err) {
        throw err;
    }
    rows.forEach( genre => {
        console.log(genre.GenreName);
    });
});
// close the database
db.close();
```

Because of the asynchronous nature of Node, a callback function must be passed to the `all()` *method: this callback will be passed all the retrieved data in an array.*

5 Save and test the page. This time, do this using the `node` command:

```
node db-tester.js
```

No browser is needed: this program runs (and then exits) entirely in the command window/terminal.

6 Add in the following code before closing the database:

```
// only put a row at a time into memory
sql = `SELECT ArtistID,FirstName,LastName
            FROM Artists WHERE NATIONALITY=? ;`;
const params = ['France'];
db.each(sql, params, (err, artist) => {
    if (err) {
        throw err;
    }
    console.log(`${artist.FirstName} ${artist.LastName}`);
});
// close the database
db.close();
```

Instead of reading the entire table into memory (which would be very memory intensive if the table was large), the callback gets called for each record.

7 Save the file. Stop the previous Node process and re-run.

8 Add in one more example and test:

```
// now get just a single record
sql = `SELECT PaintingID,Title
              FROM Paintings where PaintingID=?;`;
db.get(sql, [501], (err, painting) => {
    if (err) {
        throw err;
    }
    console.log('**** ' + painting.Title);
});
// close the database
db.close();
```

When you run this, you may find that this third query finishes before the artist query because it is simpler.

Exercise 14b.4 — Creating an API using SQLite

1 Open the provided file named `api-paintings.js` and add the following:

```
app.use(express.json());
const provider = require('./scripts/painting-provider.js');

// root endpoint will retrieve all paintings
app.get("/", (req, resp) => {
    provider.retrievePaintings(req, resp);
});

// this endpoint will retrieve single painting
app.get("/:id", (req, resp) => {
    provider.retrieveSinglePainting(req, resp);
});
```

2 Open the provided `scripts/painting-provider.js` and add the following:

```
const path = require("path");
const sqlite3 = require("sqlite3").verbose();

const DB_PATH = path.join(__dirname, "../data/art.db");
const db = new sqlite3.Database(DB_PATH);
```

So you don't have so much tedious typing, the SQL statement and the conversion function to JSON has been provided.

3 Add the following function after the provided comment:

```
// Retrieve all paintings
const retrievePaintings = (req, resp) => {
    db.all(sql, [], (err, rows) => {
        if (err) {
            throw err;
        }
        console.log('getting all paintings...');
        const paintings = rows.map(row =>
convertRecordToJson(row));
        resp.json( paintings );
    });
};
```

4 Add the following function after the provided comment.

```
// retrieve just a single painting based on the id
const retrieveSinglePainting = (req, resp) => {
    let mySQL = sql + " WHERE PaintingID=?";
    db.get(mySQL, [req.params.id], (err, row) => {
        if (err) {
            throw err;
        }
        console.log('getting single painting...');
        resp.json( convertRecordToJson(row) );
    });
};
```

5 Examine the provided function `convertRecordToJson`.

If you are implementing a JSON API using data retrieved from a SQL database, you may need to manually convert the record data into the nested JSO required for the API.

6 Run `api-paintings.js` and test in the browser via these two requests:

`http://localhost:8080/290`

This will retrieve just a single painting.

`http://localhost:8080/`

This will retrieve all paintings.

WORKING WITH MONGODB

In the next section of this lab, you will be making use of MongoDB, a server-side non-SQL database. You may or may not need to install MongoDB on your development machine.

If you are using MongoDB on the cloud, for instance via MongoDB Atlas (**which is recommended for this lab**) or MongoDB installed in AWS or GCP, you will simply access it programmatically in Node. However, to import data into your MongoDB cloud instance, **you will need to do one of the following**:

- Run mongoimport from the command line (which still requires installing a local version of MongoDB).

- Install and the run MongoDB Compass, a free GUI application. **This lab recommends this approach**.

If you do wish to install MongoDB locally, we recommend installing the Community Edition. The mechanisms for installing it vary based on the operating system.

If you wish to run MongoDB locally on a Windows-based development machine, you will need to download and run the Windows installer from the MongoDB website. If you want to run MongoDB locally on a Mac, then you will have to use HomeBrew or manual download, unpack, and copy. If you want to run MongoDB on a Linux-based environment, you will likely have to run sudo commands to do so. The MongoDB website provides instructions for most Linux environments.

Exercise 14b.5 —MongoDB Atlas and Compass

1 This exercise describes how to setup a MongoDB deployment hosted on Mongo Cloud Atlas, and then use it within the MongoDB Compass application.

To do so, you will need to create an account on mongodb.com as well as download and install the MongoDB Compass application.

2 Within Atlas, you will be prompted to create a Cluster. This may require choosing a cloud provider and region. Choose which ever one you'd like, but stick with a provider and region combination that is close to your location and which has a free tier available. Stick with the M0 Sandbox, which is free. Change the name of your cluster if you wish (mine is named funwebdev-cluster).

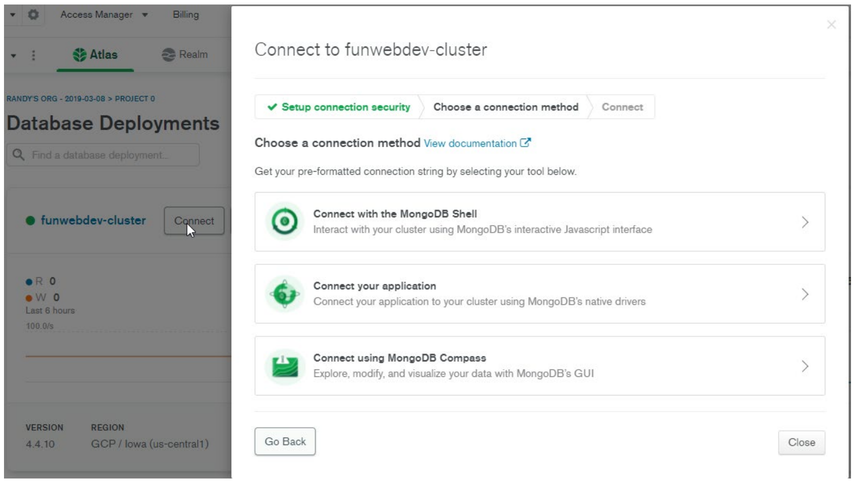
3 You now will need to create a user. On the left side of the screen, click on the Database Access option under the Security heading.

4 Click on the **Add New User** button. From the dialog, specify the user name and password, and ensure the user privilege is Read and write to any database.

5 To further protect you account, you can also use the **Network Access** option using the **Security** heading as well. This allows you to specify an IP Whitelist (that is, allowed IP addresses that are allowed to access this database). If you do not specify this, then any IP can make requests of this database.

6 Click on the **Connect** button. This will provide a dialog that will display the connection string you will need to programmatically connect to your database (see Figure 14b.1).

7 Click on the **Connect using MongoDB Compass** option. From the resulting dialog, copy the connection string.



Connect to funwebdev-cluster

Choose a connection method

Get your pre-formatted connection string by selecting your tool below.

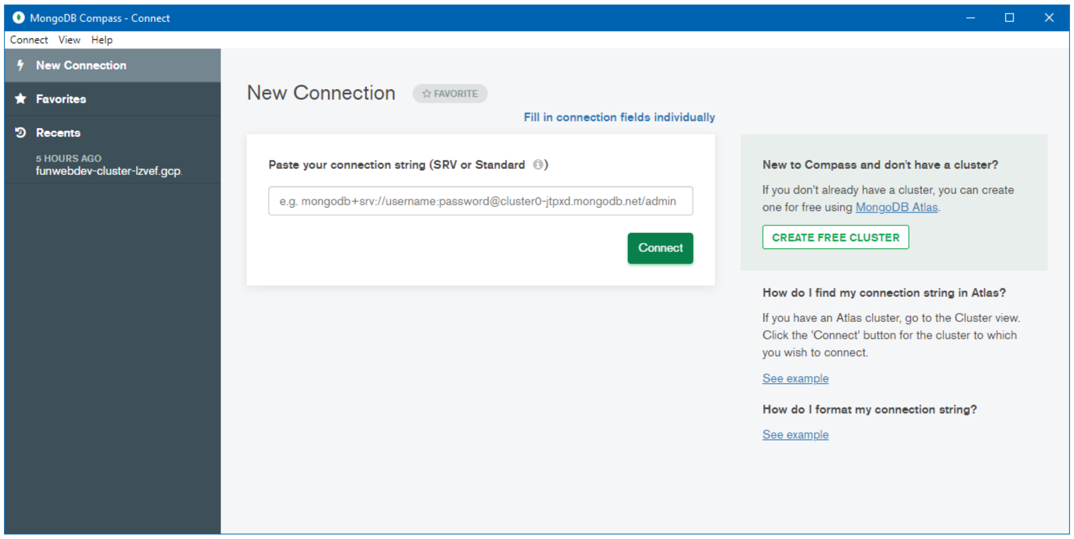
Connect with the MongoDB Shell
Interact with your cluster using MongoDB's interactive Javascript interface

Connect your application
Connect your application to your cluster using MongoDB's native drivers

Connect using MongoDB Compass
Explore, modify, and visualize your data with MongoDB's GUI

Figure 14b.1 – Getting connection information from MongoDB Cloud Atlas

8 Switch to Compass, and use the New Connection command. It may tell you that it has detected the connection string on the clipboard. If not, simply paste your connection string into the textbox and click Connect button (see Figure 14b.2).



Paste your connection string (SRV or Standard)

e.g. mongodb+srv://username:password@cluster0-jtpxd.mongodb.net/admin

New to Compass and don't have a cluster?

If you don't already have a cluster, you can create one for free using MongoDB Atlas.

How do I find my connection string in Atlas?

If you have an Atlas cluster, go to the Cluster view. Click the 'Connect' button for the cluster to which you wish to connect.

How do I format my connection string?

Figure 14b.2 – Connecting to cluster using MongoDB Compass

9 Use the **Create Database** button to add a database to your cluster. For my account, I have just the one database (called funwebdev); that database contains all the different data collections (e.g., art, travel, books, movies, users, stocks, etc) I use in the book.

10 From the dialog, specify a database and a collection name (a database must always have at least one collection). For the collection (you'll add more later), name it `books`.

11 Click on the **Add Data** button and choose the **Import File** option. Select the `books.json` file and click **Import**. When it is finished, you will be able to perform queries (which you will do below in Exercise 14b.6).

12 Add collections (and import the data) for travel images, users, and movies.

Exercise 14b.6 — Querying Data Using Compass

1 This exercise is only for those who have installed Compass (that is, completed Exercises 14b.5).

Select the books collection from the left-side of the Compass window. This should display all the data in the collection. You can then use the filter box to query the data.

2 In the filter box, enter the following then click the Find button (or press Enter):

```
{id: 587}
```

This displays a single record.

3 In the filter box, find the following:

```
{"category.secondary" : "Calculus"}
```

This finds all books whose secondary category field is equal to Calculus. Notice that you reference objects-within-objects via dot notation and that such a compound name needs to be enclosed in quotes.

4 Find the following:

```
{ "production.pages": {$gte: 950}}
```

This uses a MongoDB comparison operator to retrieve all books whose page count is greater or equal to 950.

5 Click on the More Options button, and enter the following in the Sort area:

```
{title: 1}
```

This sorts the results of the find on the title field (See Figure 14b.3). You can see

6 Click on the Reset button and then find the following:

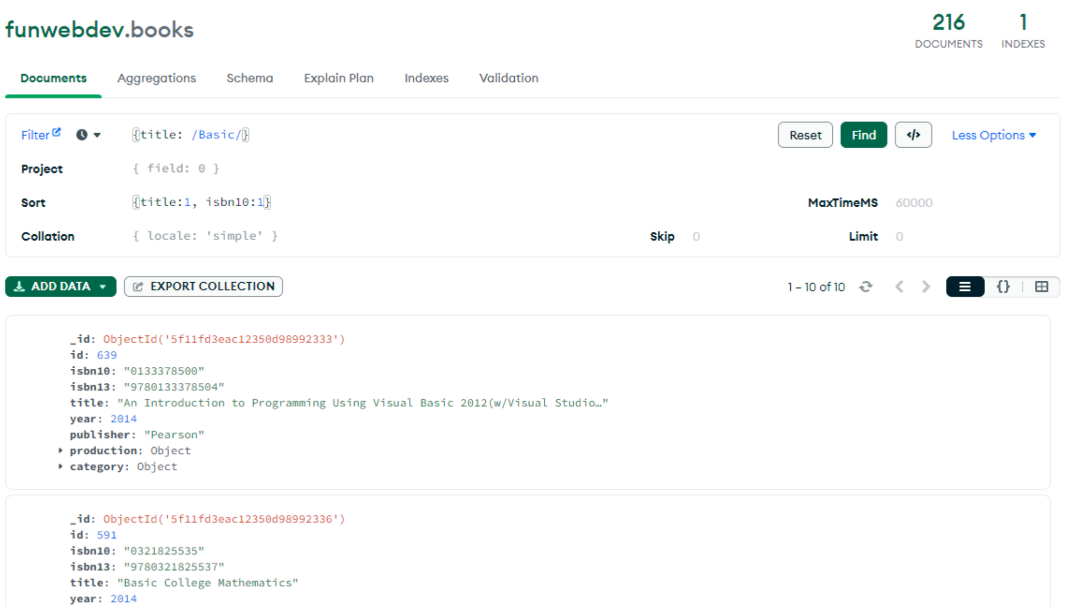
```
{title: /Basic/}
```

This uses a regular expression to do a query equivalent to SQL LIKE.

7 Click on the Options button, and enter the following in the Project (for projection) area:

```
{title:1, isbn10:1}
```

This specifies the projection (i.e., the fields you wish to return/display).



```
Filter      {title: /Basic/}
Project     { field: 0 }
Sort        {title:1, isbn10:1}                      MaxTimeMS  60000
Collation   { locale: 'simple' }       Skip  0       Limit      0
```

```
_id: ObjectId('5f11fd3eac12350d98992333')
id: 639
isbn10: "0133378500"
isbn13: "9780133378504"
title: "An Introduction to Programming Using Visual Basic 2012(w/Visual Studio…"
year: 2014
publisher: "Pearson"
production: Object
category: Object
```

```
_id: ObjectId('5f11fd3eac12350d98992336')
id: 591
isbn10: "0321825535"
isbn13: "9780321825537"
title: "Basic College Mathematics"
year: 2014
```

Figure 14b.3 – Querying a collection using MongoDB Compass

These two exercises provide just a small sampling of the types of queries one can run in MongoDB. Querying gets a lot more complex when one wants to use aggregating functions or modifying data. This lab is only intended to provide some relatively simple example uses of MongoDB in a Node application.

Exercise 14b.7 — Setting Up MongoDB Locally

1. This exercise describes how to setup a MongoDB deployment hosted on your local development machine. **If you are not going to run MongoDB locally, then skip this exercise.**

   After you have installed it, you will also need to create a data folder for it before you start MongoDB. By default, MongoDB wants to use the `/data/db` folder for its data. If you don't have this folder, create it. You may need to set read/write permissions on this folder.

2. To use MongoDB, the database process must be running. You can start this process using the following command within a terminal or command prompt.

   ```
   ./mongod
   ```

   This process must remain running in its own terminal for the duration of the time you use MongoDB. Depending on your environment, the leading ./ might not be necessary.

3. You have been provided with a three JSON data files. To import data into MongoDB, you need to use the `mongoimport` program via the following command (my word processor must break this long line into multiple lines; **however this must be typed in as a single line**):

   ```
   ./mongoimport --db funwebdev --collection books --file
   books.json --jsonArray
   ```

This instructs the program to create a database named `funwebdev`*, and a collection named* `books` *and populate that collection from the file* `books.json` *(which has been provided for you).*

4 Import the other file using the following command:

```
./mongoimport --db funwebdev --collection images
     --file travel-images.json --jsonArray
```

5 Import the other file using the following command:

```
./mongoimport --db funwebdev --collection movies
     --file movies.json -jsonArray
```

6 Examine these json files to see the structure of the data you just imported.

Exercise 14b.8 — Querying Data Using the Mongo Shell

1 **This exercise is only for those who have installed Mongo on your local machine** (that is, completed Exercise 14b.7).

In a separate terminal window, run the Mongo shell via the following command:

```
./mongo
```

2 Enter the following command into the Mongo shell:

```
db
```

This will display the current database, which should return test*, the default database.*

3 Enter the following commands into the Mongo shell:

```
show dbs
use funwebdev
```

The first command displays a list of available databases, while the second makes funwebdev the current database.

4 Enter the following command:

```
show collections
```

A database in MongoDB is composed of collections, which are somewhat analogous to tables in a relational database.

5 Enter the following command:

```
db.books.find()
```

This displays all the data in the books collection. MongoDB uses JavaScript as its query language, not SQL.

6 Enter the following:

```
db.books.find({id: 587})
```

This displays a single record.

7 Enter the following:

```
db.books.find({id: 587}).pretty()
```

This displays a single record formatted more nicely. The equivalent of a SQL WHERE clause is specified via a JavaScript object literal representing the search string.

Notice that unlike a table record in a relational database, a Mongo document is like a JavaScript literal in that it can be hierarchical.

8 Enter the following:

```
db.books.find({"category.secondary" : "Calculus"}).pretty()
```

This finds all books whose secondary category field is equal to Calculus. Notice that you reference objects-within-objects via dot notation and that such a compound name needs to be enclosed in quotes.

9 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}})
```

This uses a MongoDB comparison operator to retrieve all books whose page count is greater or equal to 950.

10 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}}).count()
```

This puts the result through the count() function (and should return the number 6).

11 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950}}).sort({title:
1})
```

This sorts the results of the find on the title field.

12 Enter the following:

```
db.books.find({ "production.pages": {$gte: 950},
                "category.main": "Mathematics" })
```

This adds another criteria (equivalent to an SQL AND).

13 Enter the following:

```
db.books.find({title: /Basic/})
```

This uses a regular expression to do a query equivalent to SQL LIKE.

14 Enter the following:

```
db.books.find({title: /Basic/}, { title:1, isbn10:1})
```

Here you have added a second parameter to find(), in which you specify the projection (i.e., the fields you wish to return/display).

15 Enter the following:

```
exit
```

USING MONGODB IN NODE

There are several API libraries for accessing MongoDB data in Node. This lab you uses the Mongoose package, which uses a lightweight ORM (Object-Relational Mapping) approach. Since Mongoose is an ORM, you will interact with the database by defining a schema rather than run MongoDB queries.

Exercise 14b.9 — TESTING MONGOOSE

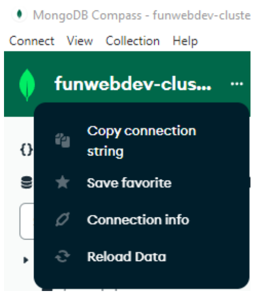
1 Enter the following commands:

```
npm install mongoose
npm install dotenv
```

You will need to specify details about your MongoDB configuration. Rather than hard coding this information in your programming code, a better idea is to put them into a separate configuration file. The dotenv package provides a mechanism for doing this.

2 You will need connection information for your database, which will vary depending on whether your using MongoDB locally or on the cloud.

If using MongoDB on Atlas and are connected, click on the … button next to the cluster name and select **Copy connection string** from the pop-up menu.



If using MongoDB locally, then the connection string will be:
mongodb://localhost:27017/funwebdev

3 Create a text file named `.env` (nothing before the dot), add the following information to it, then save. Note, you may need to modify this URL based on your MongoDB installation. If you are connecting to a cloud database, then you will need to supply the password you made in Exercise 14b.6, step 4.

```
MONGO_URL=[replace with connection string from step 2]
PORT=8080
```

It is common to include this .env file in the .gitignore file as well so that sensitive configuration details don't get uploaded to any public github repo.

4 Create a new file named `book-server.js`. Add the following content to it.

```
require('dotenv').config();
console.log(process.env.MONGO_URL);
```

5 Test by running `node book-server.js`.

This should display the MONGO URL created in step 3.

6 Edit `book-server.js` and enter the following content.

```
const mongoose = require('mongoose');
const opt = {
    useUnifiedTopology: true,
    useNewUrlParser: true
};
console.log("starting to connect to mongo ...");
mongoose.connect(process.env.MONGO_URL, opt);

const db = mongoose.connection;
db.on('error',console.error.bind(console,'connection error:'));
db.once('open', () => {
   console.log("connected to mongo");
});
```

7 Test this to make sure you can connect to the database.

If it works, you should see the "connected to mongo" message. With my free MongoDB Atlas account, the connection message appears after around 20-30 seconds.

Exercise 14b.10 — Modularizing the Code

1 Make a copy of `book-server.js` and call it `book-server-backup1.js`.

2 Create two new folders in your project, one named `models` and the other named `handlers`.

3 In the `handlers` folder, create a new file named `dataConnector.js` and move/cut the following content from `book-server.js` (i.e., from steps 4 and 6 of Exercise 14b.5) into it.

```
require('dotenv').config();
const mongoose = require('mongoose');

const connect = () => {
   const opt = {
      useUnifiedTopology: true,
      useNewUrlParser: true,
      dbName: 'your db name here'
   };
   console.log("starting to connect to mongo ...");
   mongoose.connect(process.env.MONGO_URL, opt);
   const db = mongoose.connection;
   db.on('error',console.error.bind(console,'connection error:'));
   db.once('open', function callback () {
      console.log("connected to mongo");
   });
};
```

You will need to supply the name of your database (from Exercise 14b.5, step 9). You can see your databases in Compass by clicking on Local at the top of the left-side of the window.

4 Add the following code to the end of `dataConnector.js`.

```
module.exports = {
   connect
};
```

5 Modify `book-server.js` as follows (this is all the code in the file: the other content has moved to `dataConnector.js`):

```
require('dotenv').config();

const express = require('express');
const app = express();

// create connection to database
require('./handlers/dataConnector.js').connect();


const port = process.env.port;
app.listen(port, () => {
    console.log("Server running at port= " + port);
});
```

6 Test. If everything works, you should see the appropriate console message (eventually).

Exercise 14b.11 — Creating the Model

1 In the `models` folder, create a new file named `Book.js` and add the following content:

```
const mongoose = require('mongoose');
// define a schema that maps to the structure of the data in MongoDB
const bookSchema = new mongoose.Schema({
   id: Number,
   isbn10: String,
   isbn13: String,
   title: String,
   year: Number,
   publisher: String,
   production: {
       status: String,
       binding: String,
       size: String,
       pages: Number,
       instock: Date
   },
   category: {
     main: String,
     secondary: String
   }
 });
 module.exports = mongoose.model('Book', bookSchema);
```

Note the name of the model, by default, must be a singular version of the plural of the collection. In our example, the MongoDB collection name is books; thus the model name must be Book.

2 Optionally, you can specify the name of the collection (which you need to do it the collection name isn't a singular version of the plural of the collection), by adding the collection name as the third parameter, as shown in the following:

```
module.exports = mongoose.model('Book', bookSchema, 'books');
```

3 Modify `book-server.js` as follows:

```
const app = express();
// get our data model
const Book = require('./models/Book');
```

What will we do with this Book object? We will pass it to the route handlers in the next exercise.

Exercise 14b.12 — Implementing Mongoose-Based API

1 Remind yourself of the structure of each book item by examining `single-book.json`.

2 In the `handlers` folder, create a new file named `bookRouter.js`.

3 Add in the following route:

```
//  handle GET requests for [domain]/api/books - return all books
const handleAllBooks = (app, Book) => {
   app.get('/api/books', req,resp) => {
     // use mongoose to retrieve all books from Mongo
     Book.find()
       .then((data) => {
         resp.json(data);
       })
       .catch((err) => {
         resp.json({ message: "Unable to connect to books" });
      });
   });
};
```

4 Export the handler by adding the following to the end of the file:

```
module.exports = {
   handleAllBooks
};
```

5 Add in the necessary "wiring" for express in `book-server.js` as well as the new route handler:

```
const Book = require('./models/Book');
// tell node to use json and HTTP header features in body-parser
app.use(express.urlencoded({extended: true}));
// use the route handlers
const bookRouter = require('./handlers/bookRouter.js');
bookRouter.handleAllBooks(app, Book);
```

6 Test by making the following request (your domain may be different if not running locally):

```
http://localhost:8080/api/books
```

This should display all the books in JSON format.

7 Add the following route to `bookRouter.js`.

```
// handle requests for specific book: e.g., /api/books/0321886518
const handleSingleBook = (app, Book) => {
  app.get("/api/books/:isbn", (req, resp) => {
    Book.find({ isbn10: req.params.isbn })
      .then((data) => {
        resp.json(data);
      })
      .catch((err) => {
        resp.json({ message: "Unable to connect to books" });
      });
  });
};
```

8 Export the handler by adding the following to the end of the file:

```
module.exports = {
   handleAllBooks,
   handleSingleBook
};
```

9 Add the handler in `book-server.js`.

```
const bookRouter = require('./handlers/bookRouter.js');
bookRouter.handleAllBooks(app, Book);
bookRouter.handleSingleBook(app, Book);
```

10 Test by making the following request (note: your domain may be different):

```
http://localhost:8080/api/books/0321886518
```

This should display all the data for just a single book.

11 Add the following route to `bookRouter.js`.

```
// handle requests for books with specific page ranges:
// e.g., [domain]/api/books/pages/500/600
const handleBooksByPageRange = (app, Book) => {
  app.get("/api/books/pages/:min/:max", (req, resp) => {
    Book.find()
      .where("production.pages")
      .gt(req.params.min)
      .lt(req.params.max)
      .sort({ title: 1 })
      .select("title isbn10")
      .exec()
      .then((data) => {
        resp.json(data);
      })
      .catch((err) => {
        resp.json({ message: "Unable to connect to books" });
      });
  });
};
```

This shows how Mongoose can be used to construct a more complex MongoDB query.

12 Export the handler by adding the following to the end of the file:

```
module.exports = {
   handleAllBooks,
   handleSingleBook,
   handleBooksByPageRange
};
```

13 Add the handler in `book-server.js`.

```
const bookRouter = require('./handlers/bookRouter.js');
bookRouter.handleAllBooks(app, Book);
bookRouter.handleSingleBook(app, Book);
bookRouter.handleBooksByPageRange(app, Book);
```

14 Test via:

```
http://localhost:8080/api/books/pages/200/300
```

This should display all the books whose page count is between 200 and 300.

Test Your Knowledge #1

Back in Exercise 14b.5, you imported three JSON files into mongodb. In this Test Your Knowledge, you will work with the second file, `travel-images.json`.

1 Create a new file in the `models` folder named `Image.js`. Using `Book.js` as your guide, define a model for the images collection; the file `single-image.json` can help you define the schema for this collection.

2 Create a new file in the `handlers` folder named `imageRouter.js`. Using `bookHandler.js` as your guide, define the following routes in this file:

- retrieve all images (e.g. path `/api/images/`)
- retrieve just a single image with a specific image id (e.g. path `/api/images/1`)
- retrieve all images from a specific city (e.g., path `/api/images/city/Calgary`). To make the `find()` case insensitive, you can use a regular expression:
  ```
    find({'location.city': new RegExp(city,'i')}, (err,data) =>
  {…})
  ```
- retrieve all images from a specific country name (e.g., path `/api/images/country/canada`)

3 Create a new file named `image-server.js` using `book-server.js` as your guide but using the model and handlers created in the previous two steps.

4 Be sure to test all four routes.

Exercise 14b.13 — Making Use of Express Middleware

1. Modify `book-server.js` as follows.

```
// create an express app
const app = express();

// serves up static files from the public folder.
app.use(express.static('public'));
// also add a path to static
app.use('/static', express.static('public'));
```

2. Modify `book-server.js` as follows.

```
app.use(function (req, res, next) {
    res.status(404).send("Sorry can't find that!")
});
const port = process.env.port;
app.listen(port, () => {
    console.log("Server running at port= " + port);
});
```

3. Test using the following requests.

```
http://localhost:8080/data-entry.html
http://localhost:8080/static/data-entry.html
http://localhost:8080/static/sdkjfh
http://localhost:8080/static/book-images/0132145375.jpg
http://localhost:8080/api/books
http://localhost:8080/static/abcde.jpg
```

Our server now checks in the public folder for any requested resource. If it finds it, then it serves it.

Now that we have the static file middleware working, we can perform one last MongoDB task: saving data.

Exercise 14b.14 — Saving Data in MongoDB

1. Examine the form by requesting `http://localhost:8080/data-entry.html`.

2. Edit `data-entry.html` (it is in the `public` folder) as follows.

```
<form method="post" action="/api/create/book" class="main">
```

3 Add a new route to `bookRouter.js` as follows.

```
//  handle POST request for a new book
const handleCreateBook = (app, Book) => {
  app.route('/api/create/book')
    .post( (req,resp) => {
      // retrieve the form data from the http request
      const aBook =  {
            isbn10: req.body.isbn10,
            isbn13: req.body.isbn13,
            title: req.body.title,
            year: req.body.year,
            publisher: req.body.publisher,
            production: {
               pages: req.body.pages
            }
      };
      // now have mongoose add the book data
      Book.create(aBook, (err, data) => {
      // for now simply return a JSON message
        if (err) {
           resp.json({ message: 'Unable to connect to books' });
        } else {
           const msg = `New Book was saved
                                isbn=${aBook.isbn10}`;
           resp.json({ message: msg });
        }
      });
   });
};
```

4 Export the handler by adding the following to the end of the file:

```
module.exports = {
   ...
   handleCreateBook
};
```

5 Add these handlers to `book-server.js` as follows.

```
bookRouter.handleBooksByPageRange(app, Book);
bookRouter.handleAllCategories(app, Book);
bookRouter.handleCreateBook(app, Book);
```

6 Test by requesting `http://localhost:8080/data-entry.html`. Fill in the form and make the ISBN10 of the new book `1234567890`. If you run this multiple times, you will need to change the ISBN each time.

7 Verify it worked by making the following request.

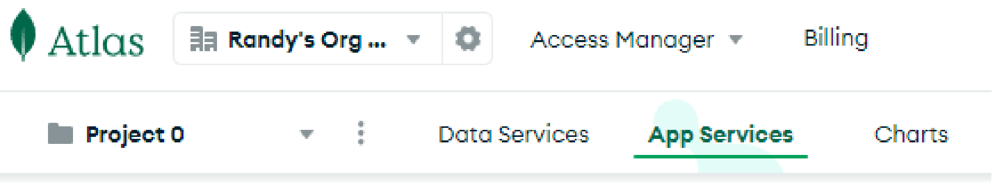
```
http://localhost:8080/api/books/1234567890
```

Serverless Approach to MongoDB

This lab has focused on using MongoDB within Node. In recent years, serverless approaches to data access have become increasingly popular. In this approach, JavaScript clients retrieve API data from cloud platforms offering Functions-as-a-Service (FaaS). This can eliminate the need to deploy and host a server environment such as PHP or Node. In 2021, MongoDB introduced the Realm FaaS for their Atlas-hosted databases (renamed to App Services). To begin, you will set up Realms so that you can access MongoDB data directly within your front-end.

Exercise 14b.15 — Setting Up App Services Functions

1 Login into your MongoDB account. Click on the App Services tab.



The first time you access this option, the system will automatically provide a series of step-by-step guides for creating your first Realm application. This interface seems to change relatively frequently so the instructions in this lab may no longer be completely accurate by the time you run them yourself. Alternately, you may need to click the Create a New App button.

2 You will be asked to supply an Application name (I simply choose the default, Application-0, but feel free to use any name), and to indicate which cluster you are using.

3 You will be need to select the database and collection using the **Rules** option on the left-side of screen under Data Access.

4 Choose the database you created back in step 9 of Exercise 14b.5) and collection (choose books). Choose the readAll preset. You may be shown an **Understanding Save and Deploy** dialog. Step through it.

5 Click on the Authentication option on left side of screen. Enable **Allow users to login anonymously**. You are now ready to create your first function.

Exercise 14b.16 — Defining App Services Functions

1 Click on the Functions option on left side of screen. Click the **Create New Function** button.

2 In the function name, enter `getAllBooks`.

3 Click on the Function Editor tab. The system provides you with the starting code for a function. Replace the provided comment and return statement with the following:

```
exports = async function(arg){
  const collection = context.services
.get("mongodb-atlas").db("funwebdev").collection("books");
  return collection.find({});
};
```

My word processor inserts a line break after services, but you can leave it one line. You will likely need to replace funwebdev with your actual database name.

3 Test your service by clicking the Run button. If it works it should display all the books in your collection.

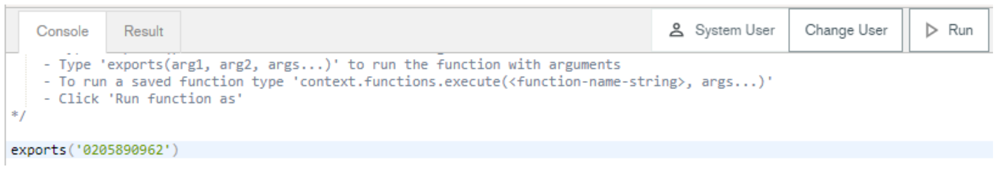
4 Click the Save Draft button.

5 Create another new function named `getBookByISBN`.

6 Click on the Function Editor tab, add the following code, and Click Save:

```
exports = async function(arg){
  const collection = context.services
.get("mongodb-atlas").db("funwebdev").collection("books");
  return collection.findOne({isbn10: arg});
};
```

7 To test, click on the Console tab, and edit the supplied exports so that it passes an ISBN (use 0205890962) to the function.

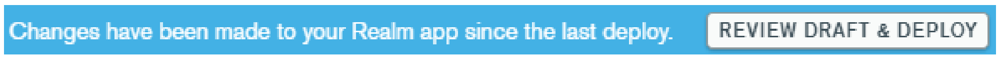


```
  - Type 'exports(arg1, arg2, args...)' to run the function with arguments
  - To run a saved function type 'context.functions.execute(<function-name-string>, args...)'
  - Click 'Run function as'
*/

exports('0205890962')
```

8 Click Clear Result button, then the Run button. It should display the requested book.

9 If it worked correctly, click the Save Draft button (if visible), then click the **Review Draft & Deploy** button.


Changes have been made to your Realm app since the last deploy.

10 Click Deploy in the dialog. You are now ready to use this function in Node or React.

Exercise 14b.17 — Setting Up React Tester

1 Assuming you have completed lab11b, you should have create-react-app installed.

Run the following command if you have npx:

```
npx create-react-app my-app
```

2 In your terminal, switch to `my-app` folder.

3 Type and run the following command:

```
npm install realm-web
```

4 Copy the contents of the `realm-src` folder in the supplied `lab14b` folder into the `src` folder of `my-app`.

5 To run and test our project, you need to switch to the `my-app` folder and run the project via:

```
npm start
```

Exercise 14b.18 — Consuming Mongo Service

1 Examine `App.js`. Notice that is defines a simple books array. You will be replacing this with data from Mongo App Services.

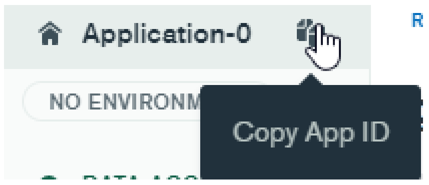
2 Add the following `import` statements to the top of `App.js`.

```
import { useEffect, useState } from 'react';
import * as Realm from "realm-web";
```

3 Comment out the existing array definition in `App.js` and replace it with the following state definition:

```
//const books = [{title:"hamlet"},{title:"macbeth"}];
const books, setBooks] = useState([]);
```

4 Copy the App ID via the icon in the mongodb Realms page:



5 Add the following `useEffect` which will invoke the Realm function you defined earlier. Replace “[your app id here]” (including brackets) with the App ID you copied in Step 4.

```
useEffect( () => {
  async function fetchData() {
    const REALM_APP_ID = "[your app id here]";
    const app = new Realm.App({ id: REALM_APP_ID });

    const credentials = Realm.Credentials.anonymous();
    try {
      const user = await app.logIn(credentials);
      const allBooks = await user.functions.getAllBooks();
      setBooks(allBooks);
    } catch(err) {
      console.error("Failed to log in to Realm", err);
    }
  }
  fetchData();

}, [])
```

6 Test. Your sample page should now display the data retrieved from the Realm function.

Instead of consuming a Node-based API, this React application is retrieving its data from a serverless function.