LAB 20d

INTERMEDIATE NODE

What You Will Learn

- How to read data from SQLite
- How to setup MongoDB
- How to import JSON data into Mongo
- How to query data in Mongo
- How to read Mongo data in Node using Mongoose
- What is middleware in Express and how to make use of it
- How to use the Pug View Engine in Express

Approximate Time

The exercises in this lab should take approximately 100 (or 1000?) 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 3, 2020

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.

PREPARING DIRECTORIES

1 This lab has additional content (SQLite files, JSON files) in two folders (named `data` and `public`) that you will need to copy/upload this folder into your eventual working folder/workspace.

Exercise 20d.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 20d.2 — Setting Up SQLite

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 -g nodemon
   ```

   This installs the Nodemon utility globally on your machine. Nodemon is a very useful utility program: it monitors your source code and whenever a file changes, it stops your server (i.e., currently running Node application) and restarts it automatically. This means you won't have to worry about remembering to break your running Node process and then restarting it every time you make a code change.

3. Enter the following commands:

   ```
   npm install -save express
   npm install -save sqlite3
   ```

4. 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.

Exercise 20d.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 `nodemon` command:

```
nodemon 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}`);
});
```

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. Because you used nodemon, the saved version is 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);
});
```

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

Exercise 20d.4 — Creating an API using SQLite

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

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

```
// 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:

```
// 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 Run using the `nodemon api-paintings.js` command

6 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.

Exercise 20d.5 — SETTING UP MONGODB

1 You may or may not need to install MongoDB on your development machine. If you are using MongoDB on the cloud (for instance via MLab or MongoDB Atlas or MongoDB installed in AWS or GCP), you will simply access it programmatically in Node.

If you are using a cloud-based development environment (for instance Cloud9), MongoDB might already installed in your workspace.

If you do wish to install MongoDB, 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.

2 If you have installed MongoDB locally, 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.

3 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.

4 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).

5 Import the other file using the following command:

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

6 Import the other file using the following command:

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

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

Exercise 20d.6 — Using the Mongo Shell

1 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
```

Exercise 20d.7 — Adding Data

1 If not already running, run the Mongo shell via the following commands:

```
./mongo
```

2 Ensure you are using the correct database via:

```
use funwebdev
```

3 Enter the following:

```
db.sales.insert({ "id": "1",  "amount": 20.00 })
db.sales.insert({ "id": "2",  "amount": 30.00 })
db.sales.insert({ "id": "3",  "amount": 40.00 })
db.sales.insert({ "id": "4",  "amount": 50.00 })
```

Notice that you can create a new collection simply by adding new objects.

4 Enter the following:

```
db.sales.find()
```

5 Enter the following:

```
db.sales.update( {"id": "3"}, { "id": "3", "amount": 50.00 })
```

6 Enter the following:

```
db.sales.remove( { "id": "4" }, { justOne: true } )
```

The above exercises illustrated just the simplest queries in MongoDB. More complicated queries using aggregation functions add a significant layer of complexity. The next exercise illustrates a few examples, but a more complete exploration is beyond the scope of this lab.

Exercise 20d.8 — Using Aggregate Functions

1 If not already running, run the Mongo shell via the following commands:

```
./mongo
```

2 Ensure you are using the correct database via:

```
use funwebdev
```

3 Enter the following:

```
db.sales.aggregate([ { $group: { _id: null,
                        total: {$sum: "$amount"}} } ])
```

While shown as two lines, you would type this in as a single line.

The aggregate function allows you to perform aggregations similar to GROUP BY in SQL. It is typically provided with a $group specifier. The `_id` is used to indicate which fields to group the aggregator on. In this case, we are not grouping but summing a value across the entire collection so it is set to null. The `$sum` indicates we want a sum on the field named `amount`.

4 Enter the following:

```
db.books.find({title: /algebra/i}).pretty()
```

Returns all books with Algebra (case insensitive) in the title.

5 Enter the following:

```
db.books.aggregate([ { $match: {title: {$regex: /Algebra/i}} } ])
```

This does the same thing.

6 Enter the following:

```
db.books.aggregate([
   { $group: { _id: "$category.main" }} ])
```

This provides a grouping on the category.main field.

7 Enter the following:

```
db.books.aggregate([
   { $group: { _id: "$category.main",
               count: {$sum: 1} }} ])
```

This adds a count of how many books there are for each category.

8 Enter the following:

```
db.books.aggregate([
   { $group: { _id: "$category.main",
               count: {$sum:1} }},
   { $sort: {_id:1} }])
```

This sorts the data by the category.main field.

Using MongoDB in Node

There are several API libraries for accessing MongoDB data in Node. In this lab, you will be making use of 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 20d.9 — Testing Mongoose

2 Enter the following commands:

```
npm install -save mongoose
npm install -save 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.

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.

```
MONGO_URL=mongodb://localhost:27017/funwebdev
```

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 `nodemon book-server.js`.

This should display the 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
};
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");
});
```

7 Test this to make sure you can connect to the database. Remember than Mongo server has to be running in its own terminal window first (see step 3, Exercise 20d.5)

If it works, you should see the "connected to mongo" message.

Exercise 20d.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 20d.5) into it.

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

const connect = () => {
   const opt = {
        useUnifiedTopology: true,
        useNewUrlParser: true
   };
   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");
   });
};
```

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`):

```
const express = require('express');
const parser = require('body-parser');

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

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

let port = 8080;
app.listen(port, function () {
    console.log("Server running at port= " + port);
});
```

6 Verify this compiles.

Exercise 20d.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 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 20d.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.route('/api/books')
      .get(function (req,resp) {
         // use mongoose to retrieve all books from Mongo
         Book.find({}, function(err, data) {
            if (err) {
               resp.json({ message: 'Unable to connect to books' });
            } else {
               // return JSON retrieved by Mongo as response
               resp.json(data);
            }
         });
      });
};
```

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(parser.json());
app.use(parser.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.route('/api/books/:isbn')
      .get(function (req,resp) {
         Book.find({isbn10: req.params.isbn}, (err, data) => {
            if (err) {
                resp.json({ message: 'Book not found' });
            } else {
                resp.json(data);
            }
         });
      });
};
```

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.route('/api/books/pages/:min/:max')
   .get(function (req,resp) {
      Book.find().where('production.pages')
            .gt(req.params.min)
            .lt(req.params.max)
            .sort({ title: 1})
            .select('title isbn10')
            .exec( function(err, data) {
                    if (err) {
                      resp.json({ message: 'Books not found' });
                    } else {
                      resp.json(data);
                    }
             });
   });
};
```

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.

Exercise 20d.13 — Aggregate functions in Mongoose

1 You were introduced to aggregate functions back in Exercise 20d.8. You can use aggregate functions as well in Mongoose. Add the following route to `bookRouter.js`.

```
//  requests for [domain]/api/categories  e.g. return all categories
const handleAllCategories = (app, Book) => {
  app.route('/api/categories')
    .get(function (req,resp) {
       // use an aggregrate function for this query
        Book.aggregate([
          { $group: {_id: "$category.main", count: {$sum:1}} },
          { $sort: {_id:1} }
        ], (err, data) => {
             if (err) {
               resp.json({ message: 'Unable to connect to books' });
             } else {
               resp.json(data);
             }
       });
   });
};
```

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

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

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

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

4 Test via `http://localhost:8080/api/categories`

Test Your Knowledge #1

Back in Exercise 20d.5, you imported three JSON files into mongodb. In this Test Your Knowledge, you will work with the second file, `travel-images.json`, which you imported into a collection named `images`.

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.

Specifying Middleware in Express

Middleware in Express refers to functions that have access to the `request` and `response` objects and which work as a pipeline, that is, each middleware function passes on control to the next middleware function. You have already made use of Express middleware in the following two lines:

```
app.use(parser.json());
app.use(parser.urlencoded({extended: true}));
```

Both `parser.json()` and `parser.urlencoded()` are functions and get executed with every request. The express function `use()` is how we specify which middleware functions are to be called with every request. So far, you have only used Express functions as middleware, but you can also add in your own (or others) application-specific middleware functions. For instance, a common middleware function would be some type of authentication and/or authorization check on each request.

Exercise 20d.14 — Making Use of Express Middleware

1 Make a copy of `book-server.js` and call it `book-server-backup2.js`.

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

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

/* --- middleware section --- */

// 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'));

// convert raw requests into usable data
app.use(parser.json());
app.use(parser.urlencoded({extended: true}));
```

3 Test using the following requests.

```
http://localhost:8080/data-entry.html
http://localhost:8080/static/data-entry.html
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.

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

```
app.use(parser.urlencoded({extended: true}));
// customize the 404 error with our own middleware function
app.use(function (req, res, next) {
    res.status(404).send("Sorry can't find that!")
});
```

5 Test using the following requests.

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

You will notice that the books API no longer works. Notice that the middleware function in step 4 is unconditional: it sends a 404 status if no other handler has been invoked.

6 Move the code from step 4 to *after* the route handlers.

```
app.use(parser.urlencoded({extended: true}));
// set up route handlers
bookRouter.handleAllBooks(app, Book);
bookRouter.handleSingleBook(app, Book);
bookRouter.handleBooksByPageRange(app, Book);
bookRouter.handleAllCategories(app, Book);
// customize the 404 error with our own middleware function
app.use(function (req, res, next) {
    res.status(404).send("Sorry can't find that!")
});
```

7 Test using the same requests as step 5 above.

Everything should work properly now.

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

Exercise 20d.15 — 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(function (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, function(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`. Make the ISBN10 of the new book `1234567890`.

7 Verify it worked in database by using the following commands in the mongo shell.

```
db.books.find( {isbn10: "1234567890"});
```

```
db.books.remove( {isbn10: "1234567890"}, {justOne: true});
```

Adding a View/Template Engine

In the two Node labs so far, we have used it as a file and API server. You may have wondered if Node can be used in a way similar to PHP. The answer is yes. It is possible to make use of a view engine to programmatically render HTML in Node. Perhaps the most popular of these is **Pug** (formerly called Jade).

The way a view engine works is that you create your views using some specialized format that contains presentation information plus JavaScript coding (this file is the *template*). This template file is somewhat analogous to PHP files in that they are a blend of markup and programming).

With Pug, you specify your presentation in `.pug` files. These do not use HTML, but its own special syntax that is "converted" into HTML by the Pug view engine at run-time. Another popular alternative to Pug is EJS, which uses regular HTML with JS embedded within `<% %>` tags.

Exercise 20d.16 — Introducing Pug

1. Enter the following commands:

```
npm install -save pug
```

2. Create a new folder in your application named `views`.

   This folder is going to contain your .pug files.

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

```
/* --- middle ware section --- */

// view engine setup
app.set('views', './views');
app.set('view engine', 'pug');
```

4. Create a new file named `index.pug` within the `views` folder.

5. Add the following content to this new file.

```
doctype html
html
   head
      title= title
   body
      h1= heading
      p First paragraph
      p
         | second paragraph
      div
         | Within the div
         img(src='static/book-images/0132145375.jpg')
```

The pug format uses indentation instead of angle <> brackets to indicate the markup. You can use tabs or spaces for your indentation but must be consistent throughout.

Notice also the title= and h1= lines. This is how we insert dynamic content into the pug file (similar to `<?php =$variable ?>` in PHP).

6 Finally create a route to render the `index.pug` file within `book-server.js`.

```
// for root requests, render the index.pug view
app.get('/', function (req, res) {
    res.render('index', { title: 'Node 2 Lab',
                          heading: 'Sample Pug File' })
});
// set up route handlers
bookRouter.handleAllBooks(app, Book);
```

7 Test using the following requests.

```
http://localhost:8080/
```

This should display the rendered version of index.pug. Be sure to perform a View Source in the browser to see the HTML generated by the pug engine.

Exercise 20d.17 — Template Inheritance in Pug

1 Create a new file named `layout.pug` within the `views` folder.

2 Add the following content to this new file.

```
doctype html
html
   head
      title Node 2 Lab
      link(rel='stylesheet', href='/styles.css')
   body
      header
         h2 Book Site
      main.book
         section.sidebar
            nav
               ul
                  li
                     a(href='/') Home
                  li
                     a(href='/site/list') List
                  li About
                  li Contact Us
         section.pagecontent
            block content
      footer
         p &copy 2020
```

Notice the `block content` inside. This defines an area that will be defined within other pug files that extend/use this layout.

3 Create a new file named `list.pug` within the `views` folder. Add the following content

```
extends layout.pug

block content
   h1 List of Books
   div.booklist
      each book in bookData
         div
            a(href=`/site/book/${book.isbn10}`)
               img(src=`/book-images/${book.isbn10}.jpg` width='100')
         div
            h3= book.title
```

Notice that this file extends layout.pug. It thus mainly consists of the block content for this page. Notice also that this page contains a loop which iterates through the book data passed to this page.

4 Modify `index.pug` so it contains the following content.

```
extends layout.pug

block content
   h1 Home Page
   p More content here in the future
```

5 Now that the views have been defined, we need to define the routes that will make use of these views. Move the content created in Step 6 of Exercise 20d.16, into a handler within `bookRouter.js`, as shown below.

```
const handlePageIndex = (app, Book) => {
   app.route('/')
      .get(function (req,resp) {
         resp.render('index', { title: 'Node 2 Lab',
                                heading: 'Sample Pug File' })
      });
};
```

6 Add another route handler in `bookRouter.js` as shown below.

```
const handlePageBooks = (app, Book) => {
   app.route('/site/list')
      .get(function (req,resp) {
         Book.find({}, function(err, data) {
            if (err) {
               resp.render('error', { page: 'site/list'});
            } else {
               resp.render('list', { bookData: data });
            }
         });
      });
   };
```

7 Add these new handlers to the exports:

```
module.exports = {
   ...
   handlePageIndex,
   handlePageBooks
};
```

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

```
bookRouter.handleCreateBook(app, Book);
bookRouter.handlePageIndex(app, Book);
bookRouter.handlePageBooks(app, Book);
```

9 Test using the following requests. Be sure to test the Home and List links.

```
http://localhost:8080/
```

Exercise 20d.18 — More Pugging

1 Create a new file named `book.pug` within the `views` folder and add the following to it.

```
extends layout.pug

block content
   article.book
      div
         img(src=`/static/book-images/${bookData.isbn10}.jpg`)
      div
         h1= bookData.title
         p
            | Year:
            span= bookData.year
         p
            | Publisher:
            span= bookData.publisher
         p
            | Pages:
            span= bookData.production.pages
         p Categories: #{bookData.category.main},
#{bookData.category.secondary}
```

Note: there is no line break in the last line; there isn't enough space on page to show this as single line.

2 Add a route handler in `bookRouter.js` for this new page as shown below.

```
const handlePageSingleBook = (app, Book) => {
   app.route('/site/book/:isbn')
      .get(function (req,resp) {
         Book.find({isbn10: req.params.isbn}, (err, data) => {
               if (err) {
                  resp.render('error', { page: 'site/book'});
               } else {
                  resp.render('book', { bookData: data[0] });
               }
         });
      });
};
```

3 Add the new handler to the exports:

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

4 Add the handler to `book-server.js` as follows.

```
bookRouter.handlePageBooks(app, Book);
bookRouter.handlePageSingleBook(app, Book);
```

5 Test by going to list and then clicking on any of the images (which are links to book page).

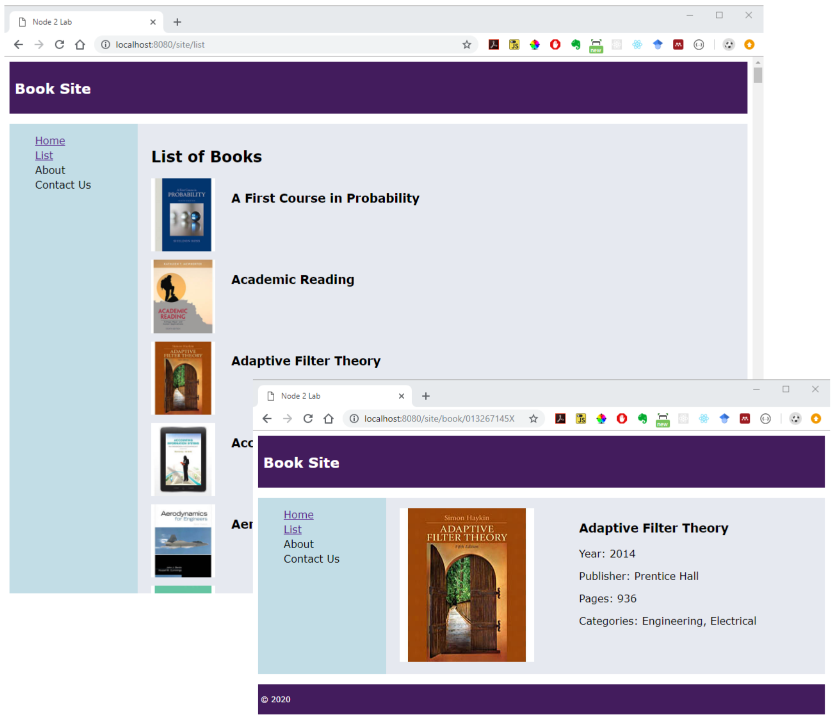


Figure 20d.1 – Sample pages built with Pug

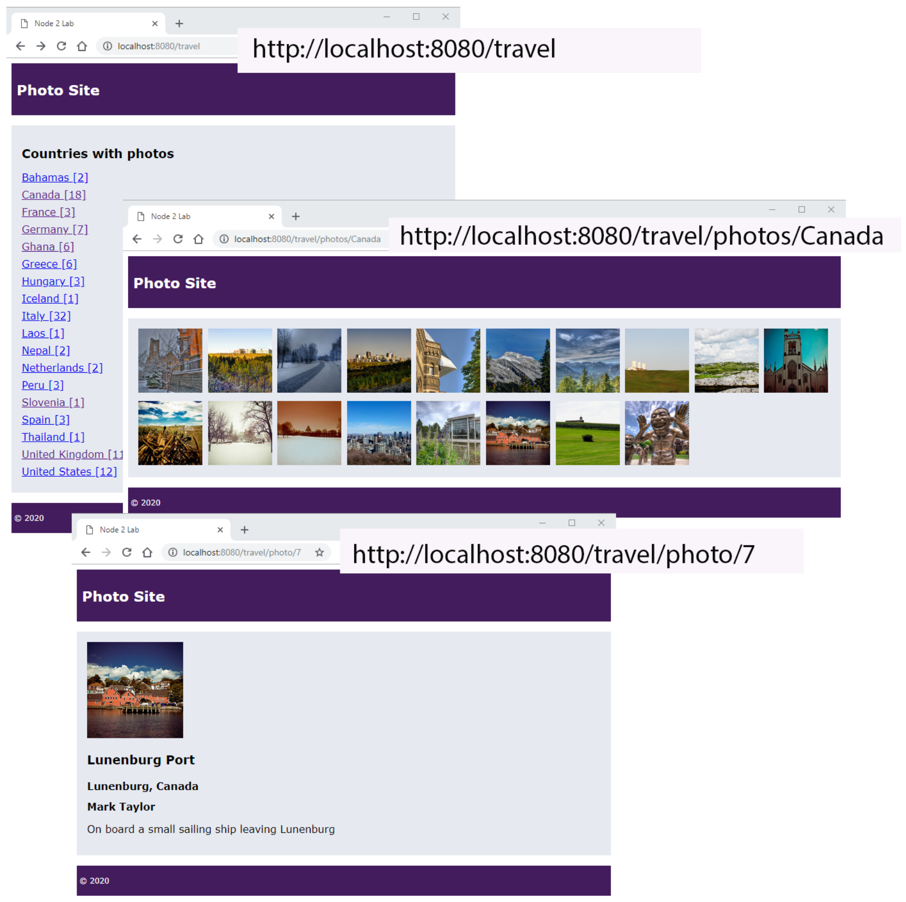
Test Your Knowledge #2

This exercise extends the finished Test Your Knowledge #1.

1 Add the following Pug views. Extend each view from a base layout view (similar to what you did in Exercise 20d.17).

- Display a list of all countries in the images collection. You will need an aggregate query similar to that in Exercise 20d.13. Make each item a link to the list of photos views.
- Display a list of photos (you have been supplied with travel images in the folder `travel-images`). The list should simply display the image. Make each image a link to the single image view (below). The list will display the photos for a specific country.
- Display the information for a single image identified by its `id`. Display the following fields: title, description, country, city, and the user first and last names.

2 Simply make use of the supplied CSS in the `styles.css` file (look towards the bottom third of the file to see the styles associated with this travel photo case.


Node 2 Lab
localhost:8080/travel
Node 2 Lab
localhost:8080/travel/photos/Canada
Node 2 Lab
localhost:8080/travel/photo/7

http://localhost:8080/travel

Photo Site

Countries with photos

Bahamas [2]
Canada [18]
France [3]
Germany [7]
Ghana [6]
Greece [6]
Hungary [3]
Iceland [1]
Italy [32]
Laos [1]
Nepal [2]
Netherlands [2]
Peru [3]
Slovenia [1]
Spain [3]
Thailand [1]
United Kingdom [1
United States [12]

© 2020

http://localhost:8080/travel/photos/Canada

Photo Site

© 2020

http://localhost:8080/travel/photo/7

Photo Site

Lunenburg Port

Lunenburg, Canada

Mark Taylor

On board a small sailing ship leaving Lunenburg

© 2020

Figure 20d.2 – Finished Test Your Knowledge