Objective:

To setup the Application with a MongoDB Container and design the Schema to be used to create the Data Model.

Supplemental Material:

Docker Compose file Reference:	https://docs.docker.com/compose/compose-file/
Mongoose Documentation:	http://mongoosejs.com/docs/guide.html
Mongoose Schema Types:	http://mongoosejs.com/docs/schematypes.html
Mongoose Validation:	http://mongoosejs.com/docs/validation.html

Preparation:

Background:

In the last lab, you moved the data that was hard-coded in your View Templates back a step to the Controllers of the MVC Data Flow. The last step of this Data Migration Process is to move the data to where it belongs, in the Model.

The advantage of doing this Data Migration Process is it solidifies the data and its format you will need for your Application; in other words, the data structures.

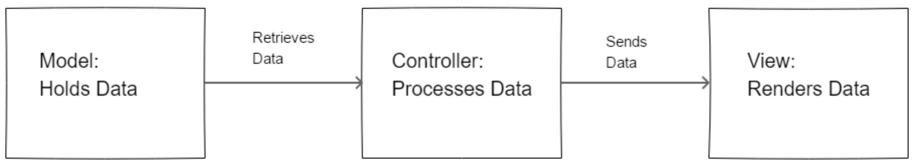

Figure 1: MVC Data Flow

Mongoose and MongoDB:

Mongoose is an Object-Document Mapper (ODM). In other words, it provides a way to map your JavaScript Objects to a document-based database like MongoDB.

First some terminology:

- **Document** - each entry in a MongoDB database is called a document. It is akin to a row in a relational database.

- **Collection** - A group of documents in a MongoDB database is called a collection. It is akin to all the rows in a table in a relational database.

- **Schema** - A schema defines the structure of a document. It defines constraints on what the data in the document should be. It is akin to a class in OOP.

- **Path** - A path is an individual data entity inside the Schema. it is akin to the properties and methods in a class definition in OOP. A Schema will contain multiple paths.

- **Model** - The model is the compiled version of the Schema.

If you think back to the project template for ICS 199, it had a very simple Schema:

Data Element (Column)	Data Type
id	int
first_name	VARCHAR(45)
last_name	VARCHAR(45)

Figure 2: The Schema for the ICS 199 Project Template

A Mongoose Schema with the same data would look like:
```
{
  first_name: { type: String },
  last_name: { type: String }
}
```

As you can see, the entire Schema is a JavaScript object called the *Schema object*. This object has properties whose values are also objects, called *Property objects*. The type is the type of data. **There are eight Schema data types you can use in Mongoose: String, Number, Date, Boolean, Buffer (for binary data like images), Mixed (any data type), Array, ObjectId**. You can read more about Schema types at http://mongoosejs.com/docs/schematypes.html.

If the only property defined is the type, a shorthand can be used:
```
{
  first_name: String,
  last_name: String
}
```

One thing you probably have noticed is that the `id` data element hasn't been defined in the Mongoose Schema. That's because it gets defined automatically. This path is called `_id` in Mongoose and has a Type of `ObjectId`. For every document, Mongoose creates this path and assigns it a unique value made up of the time, machine id, process identifiers, and a counter. For example, a Document based on the Schema declared above might look like:

```
{
  "first_name": "Bill",
  "last_name": "Gates",
  _id: ObjectId("52279effc62ca8b0c1000007")
}
```

This document is shown as JSON. MongoDB actually stores it as binary JSON (BSON).

BEFORE YOU BEGIN THE TASKS:

- As always, you should start a new git 'dev' branch from your master branch before starting the tasks in these labs.
- A good R.O.T. is to commit your work at the end of each task.
- You should run Browser-sync to automatically refresh your browser. In a bash prompt in your project's *app* folder run (assuming your container is running):

  ```
  browser-sync start --proxy localhost:3000 --no-open --reload-
  delay 2000 --files "**/*.pug, **/*.css, **/*.js"
  ```

  For Docker Toolbox users, change localhost to the IP address of your docker machine.

Tasks:

Task 1 - Adding a MongoDB Database Container

1. The first thing you obviously need to do is add a MongoDB Database to your application. Keeping the microservices architecture in mind, the Database will be created in its own container. This additional container can be created using Docker Compose. Open your project's *docker-compose.yml* file.

2. Each container in a Docker Compose file is a service. So, you will first want to add a new database service (abbreviated db). This should have the same indent as the `node:` service:

   ```
   db:
   ```

3. Next you need to specify the image you will use. As you might have guessed, there is an official Mongo Image available on Docker Hub: https://hub.docker.com/_/mongo/. Right now, the latest stable version available on Docker Hub is **3.4.10**. If we use this version for our local development, we need to make sure we use the same exact version with the deployment server (we'll worry about that in the next lab). Indented by two spaces below the `db:` line you added in the last step, add the following:

   ```
   image: mongo:3.4.10
   ```

4. Now you should give this container a name. Indented like the last step, add the following:

   ```
   container_name: mongo
   ```

5. You need to tell your node container to link to the mongo container. In the services section for your `node:` service, in-between ports and volumes, add the following line:

   ```
   links:
     - db
   ```

6. Save and close your *docker-compose.yml* file. Stop and remove any running containers. Then run `docker-compose up -d`. Use `docker-compose ps` to verify that both the node and mongo containers are up and running. If not, use `docker-compose logs` to debug.

Task 2 - Connecting to the MongoDB Instance

1. The next step is to make a connection to the database within your application code. First, you need to add the Mongoose module to your Application. Log into your node container:

   ```
   winpty docker exec -it node bash
   ```

   Then, execute the following:

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

   Docker Toolbox users will need to add the `--no-bin-links` option as well. The `--save` option means to save this new dependency to your *package.json* file. If you open this file up, you should see mongoose added to your dependencies. Exit out of the container.

2. Create a new empty text file and save it as *db.js* under your project's `models` folder. Add the following statement at the top of the file:

   ```
   const mongoose = require('mongoose');
   ```

   This of course loads the mongoose module and exposes its functions via the declared constant also named mongoose.

3. Next, add the following two statements:

   ```
   const dbURI = 'mongodb://mongo/<your project name>';
   mongoose.connect(dbURI, { useMongoClient: true });
   ```

   Replace `<your project name>` with your project's name. This is called the database connection string (a URI). There's more information you can put in here but for local development, this is all you need. The database has the same name as your project. The next statement opens a connection to it. The JavaScript object passed in as the second argument means to use mongoose's latest connection logic. It prevents getting a deprecated warning on connect.

4. Now you need to add some event listening code. When mongoose connects to a database, it fires events that you can listen for. Add the following code in *db.js*:

   ```
   mongoose.connection.on('connected', function() {
       console.log('Mongoose connected to ' + dbURI);
   });
   ```

   `connection` is an object that represents the connection to the database. This object emits events. The `on` method allows a callback to be defined for an associated event. The first argument to `on` is the name of the event. The second argument is the callback - the function to be executed when this event occurs. In this case, the callback is an anonymous function which prints out a 'connected' message to the console.

5. Add the following code to *db.js*:

```
mongoose.connection.on('error', function(err) {
    console.log('Mongoose connection error:' + err);
});

mongoose.connection.on('disconnected', function() {
    console.log('Mongoose disconnected');
});
```

These define callbacks for the 'error' and 'disconnected' events. Like in the last step, they will print messages to the console. Save and close *db.js*.

6. In order to execute the code in *db.js*, you need to add it to your Application. Open your project's *app.js* file. Right **above** your *appRouter* require statement add the following:

   ```
   require('./app_server/models/db');
   ```

   This executes the code in *db.js*. You don't need to assign it to a variable because you won't be executing any functions from it in *app.js*. Save and close *app.js*.

7. Stop, remove, and bring up your containers. At the prompt, execute the following:

   ```
   docker-compose logs | grep Mongoose
   ```

   If the connection worked, you should see a message that Mongoose successfully connected to your database. Keep this command handy, as you can use it if you ever run into problems with the connection. If there is an error or disconnect, it will be shown in the log.

Task 4 - Defining A Simple Mongoose Schema

In this task, a Schema will be defined based on the example Camosun Airways Application. In the last lab, a JavaScript object was created in the Controller to hold each flight on the home page:

```
{
   origin: 'Victoria',
   destination: 'Vancouver',
   days: ['daily']
}
```

In this task, a Schema will be created for such objects. You will do something similar for your project.

1. First, a new empty file was created under the `models` folder called `flight_schema.js`.

2. In this file, the mongoose module is required:

   ```
   const mongoose = require('mongoose');
   ```

3. Next, the Schema will need to be loaded into the application. The following statement was added to the end of *db.js*:

   ```
   require('./flight_schema');
   ```

4. The first thing that has to be decided in designing the Schema is the Schema Data Type for each path. For this example, the first two paths are Strings while the last one is an Array of Strings. So, in this case, a basic Schema looks like:

```
const flightSchema = new mongoose.Schema({
    origin: String,
    destination: String,
    days: [String]
});
```

   To create a Schema you call the constructor `Schema` and pass in the Schema object. A reference to the Schema is assigned to a constant, `flightSchema` in this example. The above code was added to *flight_schema.js*.

5. Although the Schema defined in the step above works, Mongoose allows you to add Schema-level validation as discussed at http://mongoosejs.com/docs/validation.html. It is highly recommended that you take advantage of this to prevent problems with missing or malformed data. For example, the origin and destination should be required paths for `flightSchema`:

```
const flightSchema = new mongoose.Schema({
    origin: {
      type: String,
      required: true
    },
    destination: {
      type: String,
      required: true
    },
    days: [String]
});
```

   When adding additional properties besides `type`, you can no longer use the shorthand like the previous step so you have to explicitly declare the type property. The `required: true` property, which applies to any Schema type, means the path is required.

6. We can take this a step further by enumerating the possible values for origin and destination. The String Mongoose Data Type provides a property called `enum`. You can set it to an array of possible Strings that the path can have. For example, on origin:

```
origin: {
      type: String,
      required: true,
      enum: ['Victoria', 'Vancouver', 'Nanaimo']
},
```

   This means that origin can only be a String of Victoria, Vancouver, or Nanaimo. The same can be set for destination.

Make a Schema for your project. The following notes may help you:

Numbers:

The previous example didn't talk about paths with a Number type. Suppose you want to add a rating system to your application. You want it to have a default value of 0:

```
rating: {
  type: Number,
  default: 0
}
```

The default property can be used on any Schema Data Type. You can also use the `min` and `max` property with the Number Schema Data Type. For example, if the rating should be between 0 and 5 inclusive:

```
rating: {
  type: Number,
  default: 0,
  min: 0,
  max: 5
}
```

Task 5 - Adding a Subdocument

Task 4 suffices for the simple data on the Home Page. But the Flight Details page requires a lot more data. Here is the JavaScript object powering the Flight Details page for Camosun Airways with some sample data:

```
flight: {
      origin: 'Victoria',
      destination: 'Vancouver',
      days: ['daily'],
      flightTimes: [{
        depart: '7:00am',
        arrive: '7:30am',
        flightNum:1,
        Aircraft: 'Dash-8 Q400',
        Fare: '$300'
      },{
        depart: '8:00am',
        arrive: '8:30am',
        flightNum:2,
        Aircraft: 'Cessna C172',
        Fare: '$200'
      }],
      flightDates: [
        'December 12, 2017',
        'December 13, 2017',
        'December 14, 2017'
      ]
}
```

The first three properties are the same as that of the JavaScript object for the Home Page. The next property is an array of JavaScript objects. In a traditional relational database, this array could be in a separate table and you would use a join to create a query. But in a document-based database like MongoDB, you instead make them additional documents contained within the parent document - *subdocuments*. To represent this in the Schema, you need to define nested Schemas. For example, taking the `flightTimes` array and generating a Schema gives:

```
const flightTimeSchema = new mongoose.Schema({
    depart: String,
    arrive: String,
    flightNum: Number,
    Aircraft: String,
    Fare: String
});
```

If this Schema is implemented in *flight_schema.js* **before** the `flightSchema` implementation, it can be nested in it:

```
const flightSchema = new mongoose.Schema({
    origin: {
      type: String,
      required: true,
      enum: ['Victoria', 'Vancouver', 'Nanaimo']
    },
    destination: {
      type: String,
      required: true,
      enum: ['Victoria', 'Vancouver', 'Nanaimo']
    },
    days: [String],
    flightTimes: [flightTimeSchema],
    flightDates: {
      type: [Date],
      default: Date(),
      min: Date(),
      max: Date(new Date().setFullYear(new Date().getFullYear() + 1))
    }
});
```

Add a nested Schema to the Schema you made in the last task for your Project.

Task 6 - Compiling the Schema into a Model

Once your Schema is complete, you need to compile it into a Model. It's actually through the model that your Application communicates with the Database. An instance of the Model will be created for each document.

1. To compile a Schema is really easy! For Camosun Airways, in the *flight_schema.js* file, the following statement was added:

   ```
   mongoose.model('flight', flightSchema);
   ```

   You pass two arguments: the first is the name of the model and the second is the Schema you want to compile. It will automatically compile any nested Schemas with it.