# Storing Data in MongoDB

## Introduction

In this assignment we are going to learn how to integrate the React and Node application we've developed so far with a MongoDB, non-relational database. First we'll learn how to install and configure MongoDB, then practice interacting with the database using a simple command line interface, and finally learn how to integrate to the database programmatically with the Mongoose library.

## Practice

In this section we're going to practice working with the MongoDB database. We'll install, configure, and integrate the database with the React and Node application. After you've had a chance to practice the basic skills presented here, you'll be asked to apply the skills to add a database to the Tweeter clone application we've been working through several assignments.

### Copy previous assignment to new assignment

Under the ***src*** directory create a new ***a9*** directory and copy the ***Practice***, ***Build***, ***reducers***, and ***services*** from previous assignments into the ***/src/a9*** so that this assignment is completely independent from other assignments. In ***/src/a9*** create an ***index.js*** file that loads the ***Practice*** and ***Build*** components as shown below. Fix any broken imports to ***services*** and ***reducers***. Confirm the new and previous assignment are still working. The new assignment should be identical to the previous assignment at the beginning of the assignment.

***/src/a9/index.js***

```
import React from "react";
import {Route} from "react-router-dom";
import Practice from "./Practice";              // import Practice component and
import Build from "./Build";                    // import Build component

const A9 = () => {                              // new component to hold new versions of Practice and Build
  return(
    <div>
      <Route path="/a9/practice" exact={true}>  // route to navigate to Practice component
        <Practice/>
      </Route>
      <Route path="/a9/twitter">                // route to navigate to Build component
        <Build/>
      </Route>
    </div>
  )
}

export default A9;
```

In ***Practice/index.js***, fix the ***Twitter*** (or ***Build***) link to point to the version for this assignment as shown below.

***/src/a9/Practice/index.js***

```
import React from "react";
import {Link} from "react-router-dom";

const Practice = () => {
  return(
    <div>
      <h1>Practice</h1>
      <Link to="/a9/twitter/home">Twitter</Link>     // should point to the Twitter version for this assignment
    ...
    </div>
  )
};
export default Practice;
```

In ***/src/App.js*** add a route to navigate to this assignment as shown below.

***/src/App.js***

```
...
<Link to="/a8/practice">A8</Link> |       // at the top of the document, but within the browser router and
<Link to="/a9/practice">A9</Link>         // container, put links to navigate to A8 and to A9
...
<Route path="/a9">                        // at the bottom, but within the browser router and container,
  <A9/>                                   // put a new route to navigate to assignment A9
</Route>
...
```

## Working with MongoDB

MongoDB is a non relational, or NoSQL, database that has become popular amongst Web developers. The database allows storing and retrieving raw data objects formatted in JSON which makes it very convenient for writing JavaScript programs that can easily parse and manipulate the JSON data format.

### Installing and configuring MongoDB

Download MongoDB for free from https://www.mongodb.com/try/download/community. Choose the latest version, your operating system and click ***Download***. On ***macOS***, unzip the file into ***/usr/local***. You should now have a directory such as ***/usr/local/mongodb-macos-x86_64-5.0.3*** (version might differ). Add the path to your ***.bash_profile*** in your home directory so that you'll be able to execute the MongoDB server and client in the ***bin*** directory under ***mongodb-macos-x86_64-5.0.3***. Your ***.bash_profile*** should look something like this:

***~/.bash_profile***

```
export PATH="$PATH:/usr/local/mongodb-macos-x86_64-5.0.3/bin"
```

If the ***.bash_profile*** does not exist in your home directory, then create it as a plain text file, but no extensions and a period in front of it. Restart your computer after configuring your ***.bash_profile***.

On ***Windows*** unzip the file into ***C:\Program Files***. To configure environment variables on ***Windows*** press the ***Windows + R*** key combination to open the ***Run*** prompt. In the ***Run*** prompt, type ***sysdm.cpl*** and press ***OK***. In the ***System Properties*** window that appears, press the ***Advanced*** tab and then the ***Environment Variables*** button. In the ***Environment Variables*** configuration window select the ***Path*** variable and press the ***Edit*** button. Copy and paste the path of the ***bin*** directory in your ***mongodb*** directory, e.g., ***"C:\Program Files\mongodb-macos-x86_64-5.0.3\bin"***. Your actual path might differ. Press ***OK*** and restart your computer.

## Creating a MongoDB database

To create a database, you must first start the MongoDB server and tell it what directory to store the files it will use to persist the database. You can choose to create a directory anywhere in your file system, but we suggest keeping it next to the directory where you created the React and Node server projects, but NOT inside anyother project directory itself, i.e., not in the React nor Node project directories. Wherever you choose to create the directory, remember its location since you will need it to start the MongoDB database. A good place to create the data directory is right above the root of the Node project directory. Create a directory called ***data*** and under the ***data*** directory, create another directory called ***db*** as shown below. Using the terminal, go to the directory right above the root directory of the Node project, and type the following commands.

```
mkdir  data
mkdir  data/db
```

Once the directories are created, we can start the database server and specify that it use the ***data/db*** directory to store its data. The server will start and printout several status messages and eventually printout the port it's listening at, e.g., ***"port": 27017***. Leave the server running while working through the assignment. You can stop it when you are done by pressing ***Ctrl+C*** or ***Ctrl+D***. You'll need to restart the server whenever you want to interact with it or run a Node project that uses the MongoDB database. To start the MongoDB database server from the terminal, use the ***mongod*** command as shown below. Use the ***dbpath*** option to specify the ***data/db*** directory. Note the ***double dash*** ("--") before the ***dbpath*** used below.

```
mongod   --dpath   data/db                                # that's a double dash before option dbpath

{"t":{"$date":"2021-11-06T12:02:21.875-04:00"},"s":"I",  "c":"NETWORK",  "id":23015,
"ctx":"listener","msg":"Listening on","attr":{"address":"127.0.0.1"}}
{"t":{"$date":"2021-11-06T12:02:21.875-04:00"},"s":"I",  "c":"NETWORK",  "id":23016,
"ctx":"listener","msg":"Waiting for connections","attr":{"port":27017,"ssl":"off"}}
```

While the MongDB server is running, *in a separate terminal window*, start the ***mongo*** client to interact with the database manually.

```
mongo                                                     # start the mongo client
```

When ***mongo*** prints a prompt, type ***show dbs*** to display the current databases. Keep track of all the commands you issue white using ***mongo***. Copy and paste the commands and output to a separate text file called ***mongo.log*** and save it to the root of your Node project. When you are done with the assignment, add and commit ***mongo.log*** so its available in your repository when you push it to github.

```
> show  dbs                                          # display all the databases
admin   0.000GB
config  0.000GB
local   0.000GB
```

The databases displayed were created by default when the database was installed. Use the ***use*** command to create a new database or select an existing database. Let's create a database called ***webdev*** where we can do our work as shown below. All subsequent commands will use this database from now on.

```
> use  webdev
switched to db webdev
```

## Inserting and retrieving data from a MongoDB database

In MongoDB data is organized into ***collections***, not ***tables*** like in relational databases. Data contained in collections are referred to as ***documents***, not ***records*** like in other databases. To insert ***documents*** into a ***collection*** in ***MongoDB*** database, use the ***insert*** function as shown below. In MongoDB documents are formatted as JSON objects. The syntax of the insert function is as follows:

```
db.<collection>.insert(<document>)
```

Use the ***insert*** function to store a movie document in the ***movies*** collection as shown below. Execute all commands in the ***mongo*** client, copying the commands and their results to a ***mongo.log*** file you'll include in your deliverables.

```
> db.movies.insert({title: 'Aliens', rating: 4.5})
WriteResult({ "nInserted" : 1 })
```

List the databases and confirm the ***webdev*** database is listed as shown below

```
> show dbs
admin          0.000GB
config         0.000GB
local          0.000GB
test           0.000GB
webdev         0.000GB
```

Insert three more movies into the ***movies*** collection using the following JSON objects.

- {title: 'Terminator', rating: 4.8}
- {title: 'Avatar', rating: 4.7}
- {title: 'Dune', rating: 4.9}

Use the ***find*** function to retrieve all the documents from the ***movies*** collection and confirm they display as shown below. Your ***_id*** values might differ.

```
> db.movies.find();
{ "_id" : ObjectId("618708ce2f9dcdbdb306a5d7"), "title" : "Aliens", "rating" : 4.5 }
{ "_id" : ObjectId("618709c12f9dcdbdb306a5d8"), "title" : "Terminator", "rating" : 4.8 }
{ "_id" : ObjectId("618709cd2f9dcdbdb306a5d9"), "title" : "Avatar", "rating" : 4.7 }
{ "_id" : ObjectId("618709d52f9dcdbdb306a5da"), "title" : "Dune", "rating" : 4.9 }
```

Note the objects stored in the database now have a primary key ***_id*** automatically added by MongoDB when they were inserted. Your ***_id*** values will differ. You can also retrieve specific documents by matching their properties. MongoDB primary keys are of type ***ObjectId***. To retrieve documents by their primary key ***_id***, you'll have to use the ***ObjectId*** datatype as shown below. Use a value for ***_id*** that's relevant to you.

```
> db.movies.find({_id: ObjectId("618708ce2f9dcdbdb306a5d7")});
{ "_id" : ObjectId("618708ce2f9dcdbdb306a5d7"), "title" : "Aliens", "rating" : 4.5 }
```

The example below retrieves a document from the ***movies*** collection whose ***title*** property is equal to ***Aliens***.

```
> db.movies.find({title: 'Aliens'});
{ "_id" : ObjectId("618708ce2f9dcdbdb306a5d7"), "title" : "Aliens", "rating" : 4.5 }
```

The results from the find function can be made easier to read by adding the ***.pretty()*** modifier after a ***find()***.

```
> db.movies.find().pretty()
{
	"_id" : ObjectId("618708ce2f9dcdbdb306a5d7"),
	"title" : "Aliens",
	"rating" : 4.5
}
{
	"_id" : ObjectId("618709c12f9dcdbdb306a5d8"),
	"title" : "Terminator",
	"rating" : 4.8
}
{
	"_id" : ObjectId("618709cd2f9dcdbdb306a5d9"),
	"title" : "Avatar",
	"rating" : 4.7
}
{
	"_id" : ObjectId("618709d52f9dcdbdb306a5da"),
	"title" : "Dune",
	"rating" : 4.9
}
```

### Updating and deleting data from a MongoDB database

The ***updateOne*** function can be used to update documents already in the database. The first argument identifies the document to update and the second argument specifies the properties to change.

```
db.<collection>.updateOne(<filter>, <update>)
```

For instance, the example below updates the ***rating*** of a movie document whose ***title*** is ***Avatar***. The example updates the rating from 4.7 to 4.9. Execute all commands in the ***mongo*** client, copying the commands and their results to a ***mongo.log*** file you'll include in your deliverables.

```
db.movies.updateOne({title: 'Avatar'}, {rating: 4.9})
```

Just like for the ***find*** function, the filter for the ***update*** can be the document's ID as shown below.

```
db.movies.updateOne({_id: ObjectId("618708ce2f9dcdbdb306a5d7")}, {rating: 4.5});
```

You can delete a document with the ***removeOne*** function as shown below. Similar to other functions, you can use any set of properties to identify the document you want to delete.

```
db.movies.deleteOne({_id: ObjectId("618708ce2f9dcdbdb306a5d7")});
```

Retrieve all the movies and confirm the output is what you expected

```
db.movies.find();
```

## Programming with a MongoDB database

In the previous section we practiced interacting manually with the MongoDB database through the ***mongo*** client. To create applications we're going to need to interact with the database programmatically with libraries such as Mongoose.

### Installing and connecting Mongoose to a MongoDB database

To use the Mongoose library, install it from the root of the Node project as shown below.

```
npm  install  mongoose
```

To connect to the database server programmatically, require the Mongoose library and then use the ***connect*** function as shown below. Copy and paste the code at the top of the ***server.js*** file.

***/server.js***

```
const mongoose = require('mongoose');                          // load the mongoose library
mongoose.connect('mongodb://localhost:27017/webdev');     // connect to the webdev database
```

## Creating a Mongoose schema and model

Mongoose ***schemas*** describe the structure of the data being stored in the database. Copy and paste the code below into ***/movies/schema.js*** to create a schema for the ***movies*** collection we worked on earlier.

***/movies/schema.js***

```
const mongoose = require('mongoose');      // load mongoose library
const schema = mongoose.Schema({           // create schema
  title: String,                           // title property of type String
  rating: Number                           // rating property of type Number
}, {collection: 'movies'});                // which collection name
module.exports = schema;                   // export schema so it can be used elsewhere
```

We can interact with MongoDB with Mongoose models that provide functions similar to the ones found in the mongo client: ***find()***, ***updateOne()***, ***removeOne()***, etc. In movies/model.js, create a Mongoose model from the movie schema as shown below. The functions provided by Mongoose models are deliberately generic because they can interact with any collection configured in the schema.

***/movies/model.js***

```
const mongoose = require('mongoose');                     // load mongoose library
const schema = require('./schema');                       // load movies schema
const model = mongoose.model('MovieModel', schema);       // create mongoose model from schema
module.exports = model;                                   // export so it can be used elsewhere
```

It is good practice to encapsulate all data access into a handful set of files such as the following ***Data Access Object*** where we'll implement all MongoDB interactions with the movies collection. Copy and paste the code into ***dao.js***. In later sections we will implement additional functions to insert, update, delete documents from the movies collection. They will all use the movies model.

***/movies/dao.js***

```
const model = require('./model');              // load model

const findAllMovies = () => model.find();      // use model's find function to implement findAllMovies

module.exports = {
  findAllMovies                                // export as an API to use in service
};
```

## Creating an API to retrieve data from MongoDB

The movie DAO can be used within a service to retrieve the MongoDB data through a RESTful API. In a new ***service.js*** file, create an HTTP GET endpoint mapped to URL pattern ***/rest/movies*** that uses the ***findAllMovies*** DAO function to retrieve all the movies from the database. In later sections we will add additional HTTP endpoints to create, update, and delete data from/to the ***movies*** collection.

***/movies/service.js***

```
const dao = require('./dao')                    // load dao

module.exports = (app) => {                     // accept express instance to create HTTP endpoints

  const findAllMovies = (req, res) =>           // handle HTTP GET request
    dao.findAllMovies()                         // use dao to retrieve all movies
      .then(movies => res.json(movies));        // respond with all movies

  app.get("/rest/movies", findAllMovies);       // listen for HTTP GET and notify findAllMovies

}
```

Towards the bottom of ***server.js***, require the movie's service and pass it an instance of the express library.

***/server.js***

```
require('./movies/service')(app);          // load the movie service and pass it an instance of express

app.listen(4000);
```

Restart the Node server application and point your browser at http://localhost:4000/rest/movies. Confirm that all the movies are displayed in the browser as a JSON array.

## Retrieving data from MongoDB from a React client application

Now that we have an HTTP GET endpoint let's use it to retrieve data from React. Let's create a simple React component that can retrieve the movies from the server and render them as list. First let's create a service file in the React project to send an HTTP GET request to ***/rest/movies*** and parse the movies from the response body. Copy and paste the code below in ***Movies/service.js***. In later sections we will add additional functions to create, delete, and update movies through the HTTP API implemented on the server.

***/src/a9/Practice/Movies/service.js***

```
const URL = 'http://localhost:4000/rest/movies';   // declare URL to service

export const findAllMovies = () =>                 // declare function to retrieve data from database
  fetch(URL)                                       // send HTTP GET to URL
    .then(response => response.json());            // parse JSON in body from server response

export default {
  findAllMovies                                    // export findAllMovies function to use elsewhere
};
```

Let's create a React component to retrieve the movies from the server, who in turn is retrieving data from MongoDB. In ***Movies/index.js***, create the following component.

***/src/a9/Practice/Movies/index.js***

```
import React, {useEffect, useState} from "react";
import service from './service';                    // import the movie service

const Movies = () => {
  const [movies, setMovies] = useState([]);         // create a local movies state variable
  useEffect(() =>                                   // on load
    service.findAllMovies()                         // retrieve all movies from database
      .then(movies => setMovies(movies)));          // save movies from server to local movies state variable
  return(
    <div>
      <h2>Movies</h2>
      <ul className="list-group">                   // render array of movies as an HTML unordered list
        {
          movies.map(movie =>                       // iterate over movies array from database
            <li key={movie._id}                     // render each movie as a list item
                className="list-group-item">
              {movie.title}                         // render movie's title
            </li>
          )
        }
      </ul>
    </div>
  )
};

export default Movies;
```

Import the new ***Movies*** component into ***Practice/index.js*** to test the new component. Restart both the server and client application and confirm movies render in the UI.

***/src/a9/Practice/index.js***

```
<h1>Practice</h1>
<Link to="/a9/twitter">Twitter</Link>
<Movies/>
```

## Deleting data from MongoDB with Mongoose

In the Node server application, in ***dao.js***, implement ***deleteMovie*** as shown below.

***/movies/dao.js***

```
const model = require('./model');
const findAllMovies = () =>
  model.find();

const deleteMovie = (id) =>            // implement function to remove a movie given its ID
  model.removeOne({_id: id});          // use mongoose model removeOne function to remove movie by ID
```

```
module.exports = {
  findAllMovies, deleteMovie           // export the function to use elsewhere
};
```

In ***service.js***, create an HTTP DELETE endpoint mapped to URL pattern ***/rest/movies/:id*** that uses the ***deleteMovie*** DAO function to remove a movie from the database as shown below.

***/movies/service.js***

```
...
const findAllMovies = (req, res) =>
  dao.findAllMovies()
    .then(movies => res.json(movies));

const deleteMovie = (req, res) =>             // handle HTTP DELETE request mapped to /rest/movies/:id
  dao.deleteMovie(req.params.id)              // remove movie from database with ID matching ID in params
    .then((status) => res.send(status));      // respond to client with status from database

app.delete("/rest/movies/:id", deleteMovie);  // register HTTP DELETE request handler
app.get("/rest/movies", findAllMovies);
...
```

In the React client application, in ***service.js***, implement the ***deleteMovie*** function to send an HTTP DELETE requests to the server to delete the movie from the database.

***/src/a9/Practice/Movies/service.js***

```
...
export const findAllMovies = () =>
  fetch(URL)
    .then(response => response.json());

export const deleteMovie = (id) =>       // implement function to delete movie by its ID
  fetch(`${URL}/${id}`, {                // send HTTP DELETE to URL including movie's ID
    method: 'DELETE',
  });

export default {
  findAllMovies, deleteMovie             // export deleteMovie function to use elewhere
};
```

In ***Movies/index.js***, implement the ***deleteMovie*** event handler as shown below. The handler should send an HTTP DELETE request to the server and then update the local array of movies. Invoke the handler when we click on a new ***Delete*** button added to each movie. The button passes the movie object to remove whose ID is passed to the service's ***deleteMovie***. When the server responds, we remove the movie from the local copy of ***movies*** and re-render the component. Restart both the server and client applications and confirm you can remove movies.

***/src/a9/Practice/Movies/index.js***

```
...
useEffect(() =>
  service.findAllMovies()
    .then(movies => setMovies(movies)), []);
const deleteMovie = (movie) =>                // handle delete movie button click
  service.deleteMovie(movie._id)              // notify server
    .then(() => setMovies(                    // after response, filter out the movie from local copy
      movies.filter(m => m !== movie)));
return(
        ...
        <li key={movie._id}
            className="list-group-item">
          <button                             // delete movie button
            className="btn btn-danger float-end"
            onClick={() => deleteMovie(movie)}>     // on click, notify handler
            Delete
          </button>
          {movie.title}
        </li>
        ...
```

## Inserting data with Mongoose

Use Mongoose model's ***create*** function to insert a new movie document into a MongoDB database. In ***dao.js*** in the Node server application, implement ***createMovie*** to insert a new movie document in the database as shown below.

***/movies/dao.js***

```
...
const createMovie = (movie) =>             // implement function to insert a movie into the database
  model.create(movie);                     // use mongoose model create function to insert new movie

module.exports = {
  findAllMovies, deleteMovie, createMovie  // export the function to use elsewhere
};
```

In ***service.js***, implement ***createMovie*** as shown below to respond to HTTP POST messages. The function parses the new movie from the ***body*** and uses the dao's ***createMovie*** function to insert the new movie into the database. The response from the database will contain the actual movie inserted into the database. We'll respond to the client with the inserted movie so that it can update the user interface accordingly.

***/movies/service.js***

```
...
const deleteMovie = (req, res) =>
  dao.deleteMovie(req.params.id)
    .then((status) => res.send(status));
```

```
const createMovie = (req, res) =>                    // handle HTTP POST request
  dao.createMovie(req.body)                          // get new movie from body and insert into database
    .then((insertedMovie) => res.json(insertedMovie)); // respond to client with inserted movie

app.post("/rest/movies", createMovie);               // register HTTP POST request handler
app.delete("/rest/movies/:id", deleteMovie);
...
```

In ***service.js***, in the React client application, implement the ***createMovie*** function to send a POST message as shown below. In the ***body*** of the message, include the new movie to insert into the database. The server will respond with the movie actually inserted. Parse the response from the body so the user interface can update with the new movie.

***/src/a9/Practice/Movies/service.js***

```
export const createMovie = (movie) =>      // implement function to create a new movie in the database
  fetch(URL, {                             // send a POST request to the URL
    method: 'POST',
    body: JSON.stringify(movie),           // convert movie object into a string and embed in request body
    headers: {                             // tell server string is formatted as JSON
      'content-type': 'application/json'
    }
  }).then(response => response.json());    // parse movie inserted into database from the response's body

export default {
  findAllMovies, deleteMovie, createMovie  // export function to use elewhere
};
```

In ***Movies/index.js***, implement a ***Create*** button to issue a command to create a new movie in the database. Handle the event in ***createMovie*** to use the service to send the new movie object. Send a default movie with a temporary title, e.g., 'New Movie' and use the response from the server to append new movie at the beginning of the list of movies cached in the client. Restart both the server and client applications and confirm you can create new movies.

***/src/a9/Practice/Movies/index.js***

```
  const createMovie = () =>                      // handle create movie button click
    service.createMovie({title: 'New Movie'})    // send POST message to server with new movie in body
      .then(actualMovie =>                       // server responds with movie inserted into database
        setMovies([                              // append movie at beginning of movies
          actualMovie, ...movies
      ]));
  return(
    <div>
      <button                                    // new button movie to create new movie
        onClick={createMovie}                    // notify createMovie when button is clicked
        className="btn btn-success float-end">
        Create
      </button>
      <h2>Movies</h2>
```

## Retrieving a single document with Mongoose

Let's now practice retrieving a single document. In ***dao.js***, implement ***findMovieById*** to retrieve a movie document from MongoDB by its primary key ID.

***/movies/dao.js***

```
...
const findMovieById = (id) =>              // implement findMovieById to
  model.findById(id);                      // use model's findById to retrieve document by primary key

module.exports = {
  findAllMovies, deleteMovie, createMovie,
  findMovieById                            // export function to use elsewhere
};
```

In ***service.js***, create an HTTP GET endpoint mapped to URL pattern ***/rest/movies/:id*** that uses the ***findMovieById*** DAO function to retrieve a movie from the database as shown below.

***/movies/service.js***

```
...
const createMovie = (req, res) =>
  dao.createMovie(req.body)
    .then((movie) => res.json(movie));

const findMovieById = (req, res) =>          // handle HTTP GET request mapped to /rest/movies/:id
  dao.findMovieById(req.params.id)           // retrieve movie from database with ID matching ID params
    .then(movie => res.json(movie));         // respond to client with movie from database

app.get("/rest/movies/:id", findMovieById);  // register HTTP DELETE request handler
app.post("/rest/movies", createMovie);
app.delete("/rest/movies/:id", deleteMovie);
app.get("/rest/movies", findAllMovies);
```

In the React client application, in ***service.js***, implement the ***findMovieById*** function to send an HTTP GET request to the server to retrieve a movie from the database by its ID.

***/src/a9/Practice/Movies/service.js***

```
...
export const findMovieById = (id) =>         // implement function to retrieve movie by its ID
  fetch(`${URL}/${id}`)                      // send HTTP GET to URL including movie's ID
    .then(response => response.json());      // parse movie from response's body

export default {
  findAllMovies, deleteMovie, createMovie,
  findMovieById                              // export function to use elsewhere
};
```

In ***Movies/index.js***, implement the ***findMovieById*** event handler as shown below. The handler should send an HTTP GET request to the server and then update a local ***movie*** state variable. Invoke the handler when we click on a new ***Edit*** button added to each movie. Use the new movie state variable to display the movie's title. In a later section we'll save any updates to the movie's title back to the server. Restart both the server and client applications and confirm the input field updates with the movie retrieved from the server when you click it's ***Edit*** button.

***/src/a9/Practice/Movies/index.js***

```
...
const Movies = () => {                                  // in the Movies component
  const [movie, setMovie] = useState({title: ''});      // add state variable to hold movie from server
  const findMovieById = (movie) =>                      // add event handler for new Edit button
    service.findMovieById(movie._id)                    // use service to fetch movie from server
      .then(movie => setMovie(movie));                  // update local movie state variable


  ...
      <h2>Movies</h2>
      <ul className="list-group">
        <li className="list-group-item">                // add a new list item at the top of the list
          <input                                        // including an input field to display
            defaultValue={movie.title}                  // the title of the movie from the server
            className="form-control"/>
        </li>
...
        {
          movies.map(movie =>                           // in the map that renders each movie ...
        <li key={movie._id}
            className="list-group-item">
          <button                                       // ... add a new Edit button to
            onClick={() => findMovieById (movie)}       // retrieve the movie from the server
            className="btn btn-warning float-end ms-2">
            Edit
          </button>
          <button … >Delete</button>
          {movie.title}
        </li>
...
```

## Updating data with Mongoose

In the Node server application, in ***dao.js***, implement ***updateMovie*** as shown below. The function accepts a movie's primary key ID, and updates for a movie document. The model's ***updateOne*** function is used to modify the movie's properties in the movie parameter for the movie that matches the ID parameter.

***/movies/dao.js***

```
const findMovieById = (id) =>
  model.findById(id);

const updateMovie = (id, movie) =>              // use the model's updateOne function to
```

```
  model.updateOne({_id: id},                    // update the movie whose ID is id
    {$set: movie});                             // update with new movie object

module.exports = {
  findAllMovies, deleteMovie, createMovie,
  findMovieById, updateMovie
};
```

In ***service.js***, create an HTTP PUT endpoint mapped to URL pattern ***/rest/movies/:id*** that uses the ***updateMovie*** DAO function to update a movie document in the database as shown below.

***/movies/service.js***

```
...
const findMovieById = (req, res) =>
  dao.findMovieById(req.params.id)
    .then(movie => res.json(movie));

const updateMovie = (req, res) =>           // handle HTTP PUT request
  dao.updateMovie(req.params.id, req.body)  // retrieve movie from body and ID from request parameter
    .then(status => res.send(status));      // send update status back to the client

app.put("/rest/movies/:id", updateMovie);   // listen for HTTP PUT request and notify handler
app.get("/rest/movies/:id", findMovieById);
...
```

In the React client application, in ***service.js***, implement the ***updateMovie*** function to send an HTTP PUT request to the server to update the movie in the database.

***/src/a9/Practice/Movies/service.js***

```
...
export const findMovieById = (id) =>
  fetch(`${URL}/${id}`)
    .then(response => response.json());

export const updateMovie = (movie) =>       // send new movie
  fetch(`${URL}/${movie._id}`, {            // include movie's ID in the URL
    method: 'PUT',                          // use HTTP PUT method for update
    body: JSON.stringify(movie),            // include movie updates in body as string
    headers: {
      'content-type': 'application/json'    // string formatted as JSON
    }
  }).then(response => response.json());     // parse status response from server as JSON

export default {
  findAllMovies, deleteMovie, createMovie,
  findMovieById, updateMovie                // export for UI component
};
```

In ***Movies/index.js***, implement ***updateMovie*** and ***saveMovie*** event handlers as shown below. The ***updateMovie*** handler updates the local ***movie*** state variable as you type a new title in the title input field. The ***saveMovie*** handler sends an HTTP PUT request to the server and then update the local array of ***movies***. Invoke the handler when we click on a new ***Save*** button. Restart both the server and client applications and confirm you can edit movies.

***/src/a9/Practice/Movies/index.js***

```
...
const Movies = () => {
  ...
  const [movie, setMovie] = useState({title: ''});
  const findMovieById = (movie) =>
    service.findMovieById(movie._id)
      .then(movie => setMovie(movie));
  const updateMovie = (event) =>                              // save text input changes
    setMovie({...movie, title: event.target.value});          // in the local movie state variable

  const saveMovie = () =>                                     // when user clicks save button
    service.updateMovie(movie)                                // send local movie state variable to server
      .then(() => setMovies(                                  // when server responds
        movies.map(m => m._id === movie._id ? movie : m)      // replace updated movie
      ));
  return(
      ...
      <li className="list-group-item">
        <button                                               // add Save button
          onClick={saveMovie}                                 // notify saveMovie when clicked
          className="btn btn-primary float-end">
          Save
        </button>
        <input
          onChange={updateMovie}                              // update local changes in movie state variable
          defaultValue={movie.title}
          className="form-control"
          style={{width: "50%"}}/>
      </li>
      ...
```

## Tweeter

The implementation of our Twitter clone thus far uses data files such as ***who.json*** and ***tweets.json*** hosted in a Node server application. Creating new tweets and profile changes are stored in server variables. The changes persist in the server's memory as long as the server is running. If the server reboots or crashes, all changes are lost. For this assignment, we will integrate the current Twitter Node application to a MongoDB database to persist changes permanently.

## Store tweets in MongoDB

Using the mongo client, insert the tweets in ***/data/tweets.json*** in the Node application into the database, in a new collection called ***tweets***. When inserting into the database, do not include the ***_id*** property so the database can pick its own value for ***_id***. For instance, here's an example of inserting the first tweet. Note that we've not inclided the ***_id*** property.

```
> db.tweets.insert({
    "topic": "Web Development",
    "userName": "ReactJS",
    "verified": false,
    "handle": "ReactJS",
    "time": "2h",
    "title": "React.js is a component based front end library that makes it very easy to build Single
Page Applications or SPAs",
    "tweet": "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu fugiat nulla pariatur.",
    "attachments": {
      "video": "unKvMC3Y1kI"
    },
    "logo-image": "../../../images/react-blue.png",
    "avatar-image": "../../../images/react-blue.png",
    "stats": {
      "comments": 123,
      "retweets": 234,
      "likes": 345
    });
WriteResult({ "nInserted" : 1 })
```

Make sure you get a message at the end confirming the insert was successful. To verify the documents are being inserted, use ***db.tweets.find().pretty()*** to retrieve the tweets from the database and display them in the console.

## Implement Tweet Schema and Model

In ***/db/tweets/tweet-schema.js***, implement a schema for the tweets stored in the database. Here's an example of what the schema might look like. Feel free to modify as needed.

***/db/tweets/tweet-schema.js***

```
const mongoose = require('mongoose');
const schema = mongoose.Schema({
  topic: String,
  posted: {type: Date, defaultValue: Date.now},
  userName: String,
  verified: {type: Boolean, defaultValue: false},
  handle: String,
  title: String,
  tweet: String,
  attachments: {
    image: String
  },
  time: String,
  "logo-image": String,
  "avatar-image": String,
  stats: {
    comments: {type: Number, defaultValue: 0},
    retweets: {type: Number, defaultValue: 0},
    likes: {type: Number, defaultValue: 0}
  }
}, {collection: "tweets"});
module.exports = schema;
```

In ***/db/tweets/tweet-model.js***, require the tweet schema and mongoose, and create a model for the tweets called ***TweetModel*** as shown below.

***/db/tweets/tweet-model.js***

```
const mongoose = require('mongoose');
const schema = require('./tweets-schema');
const model = mongoose.model('TweetModel', schema);
module.exports = model;
```

## Implement Tweet DAO and Service

In ***/db/tweets/tweet-dao.js***, implement a DAO that uses the model to implement CRUD operations to create, retrieve, update and delete tweets from the database. Here's a template to get you started:

***/db/tweets/tweet-dao.js***

```
const model = require('tweets-model');      // require the model

const findAllTweets = () => {};             // use model to retrieve all tweets
const createTweet = (tweet) => {};          // use model to insert a new tweet
const deleteTweet = (id) => {};             // use model to remove a tweet by its ID
const updateTweet = (id, tweet) => {};      // use model to update a tweet by its ID

module.exports = {                          // export all
  findAllTweets, createTweet,               // functions to
  deleteTweet, updateTweet                  // use elsewhere
};
```

Refactor ***/services/tweeter-service.js*** to use ***tweet-dao.js*** to interact with the database, instead of using the ***/data/tweets.json*** file. Here's a template of the file to get you started.

***/services/tweeter-service.js***

```
~~let tweets = require('../data/tweets.json');~~          // don't use tweets.json file anymore
const dao = require('../db/tweets/tweets-dao');    // use dao instead to retrieve from database
module.exports = (app) => {
  const findAllTweets = (req, res) =>              // reimplement findAllTweets to use dao.findAllTweets
    dao.findAllTweets()
      .then(tweets => res.json(tweets));
  const createTweet = (req, res) => { … }          // reimplement createTweet to use dao.createTweet
  const deleteTweet = (req, res) => { … }          // reimplement deleteTweet to use dao.deleteTweet
  const likeTweet = (req, res) => { … }            // reimplement likeTweet to use dao.updateTweet

  app.put('/api/tweets/:id/like', likeTweet);
  app.delete('/api/tweets/:id', deleteTweet);
  app.post('/api/tweets', createTweet);
  app.get('/api/tweets', findAllTweets);
};
```

## Challenge (required for graduates)

As a challenge, reimplement ***WhoToFollow*** and ***Profile*** to use MongoDB for permanent storage. Use the JSON files currently in the React and or Node and insert them into the MongoDB database. Create Mongoose schemas, models, data access objects, and services required to create, read, update, and delete the relevant data.

### Refactor WhoToFollow to use MongoDB

Insert the JSON objects in ***who.json*** into a collection called ***who*** in the MongoDB database. In the Node server application, create the following database related files

| ***File*** | ***Description*** |
|---|---|
| ***/db/who/who-schema.js*** | Implements Mongoose schema describing the fields and datatypes of a who object. Configures using a collection called ***who*** |
| ***/db/who/who-model.js*** | Implements a Mongoose model from the ***who-schema*** |
| ***/db/who/who-dao.js*** | Implements a DAO with functions such as ***findAllWho*** that uses the ***who-model*** to retrieve all who documents from the database |
| ***/services/who-service.js*** | Implements a service that exposes ***findAllWho*** to an HTTP GET request mapped to ***/rest/who*** |

Here's an example of what the ***who-service.js*** might look like. Feel free to make any changes you might need.

***/services/who-service.js***

```
const dao = require("../db/who/who-model.js");    // require the dao
```

```
module.exports = (app) => {                          // accept express instance
   const findAllWho = (req, res) =>                  // handle HTTP GET request for all documents
      dao.findAllWho()                               // use DAO to retrieve documents from the database
         then(who => res.json(who));                 // respond to client with all documents as JSON array
   app.get("/rest/who", findAllWho);                 // map HTTP GET URL request to handler
}
```

In the React client application, implement ***who-service.js*** to retrieve the data from the server. Refactor the ***WhoToFollowList*** component to use the new service to render the component with data from the server instead of ***who.json***.

## Refactor Profile and EditProfile components to use MongoDB

Insert a JSON object representing a profile in a collection called ***profiles*** in the MongoDB database. In the Node server application, create the following database related files

| ***File*** | ***Description*** |
|---|---|
| ***/db/profile/profile-schema.js*** | Implements Mongoose schema describing the fields and datatypes of a profile object. Configures using a collection called ***profiles*** |
| ***/db/profile/profile-model.js*** | Implements a Mongoose model from the ***profile-schema*** |
| ***/db/profile/profile-dao.js*** | Implements a DAO that implements functions such as ***findProfileById*** and ***updateProfile***, that uses the ***profile-model*** to retrieve and update a profile document from the database |
| ***/services/profile-service.js*** | Implements a service that exposes ***findProfileById*** and ***updateProfile*** to an HTTP GET and PUT requests mapped to ***/rest/profile*** and ***/rest/profile/:id*** respectively |

# Database Management Tools

MongoDB provides a collection of command-line utilities to working with MongoDB called ***MongoDB Database Tools***. The tools include scripts such as ***mongodump***, ***mongoimport***, and ***mongoexport***. Download the tools from https://www.mongodb.com/try/download/database-tools. Choose your operating system and click Download. Unzip the downloaded file into ***/usr/local*** on macOS or ***C:\Program Files*** on Windows. Add the ***bin*** directory in the tools directory to your ***PATH*** using the same process described earlier in this document and restart your computer. Using the ***mongoexport*** tool, export the collections in the ***webdev*** database we worked with this assignment. Make sure MongoDB is running in the same directory where you created the ***data/db*** directory and execute the following commands in a separate terminal window to export the collections.

```
mongoexport --collection=movies mongodb://127.0.0.1:27017 --out=movies.json
mongoexport --collection=movies mongodb://127.0.0.1:27017 --out=tweets.json
```

Export any other collections that you might have worked on in this assignment. Copy the resulting JSON files into your Node server project directory, add them to the repository, commit the files, and push to your remote git repository so the TAs can take a look at them.

## Deliverables

As a deliverable, make sure you complete the ***Practice***, ***Build*** and ***Challange*** (if graduate student) sections of this assignment. All previous assignments must be accessible from the root context of your Website. Add, commit and push all work to your repository and confirm the Website works on Netlify. Make sure your Node project includes the ***mongo.log*** file you captured while interacting with MongoDB with the mongo client, as well as the movies.json and tweets.json containing the data exported from your MongoDB..