**Overview**

**Mongoose**

Since we just learned MongoDB, we should probably connect it to a project to see it in action! The most popular way of using MongoDB with Node and Express **is not** establishing a direct connection. What we are going to do instead is use a Node module called **Mongoose**. **Mongoose is able to connect to a MongoDB database and it will allow us to give a little bit more structure to our data by providing functionality in the form of models and schemas**. Remember when we said that MongoDB is really free-form? Well, that's very helpful in some ways, but a little more structure allows us to do things with **a lot more confidence in the integrity of our data**.

Mongoose allows us to make models in the traditional MVC sense. **It acts as a layer between our application and our database**, which is a very common pattern in software development. By converting our free-form MongoDB data into models, we are able to do things like validate, add associations, and run more intricate queries more effectively.

For example: Let's assume we wanted to build a **mongoose model** for a **user collection** in our MongoDB. Go to your MongoDB command line and make sure your server is running (you can check your terminals to ensure the server is still running). Now, let's **create a new project called basic\_mongoose\_app** and set it up as instructed in the previous tabs (EJS and everything). Now let's take a look at our **server.js** file and get started. (You should create a basic app that has *express, ejs*, and *body-parser*).

make a package.json file:

yourcommandline> npm init -y

yourcommandline> npm install express --save

yourcommandline> npm install ejs --save

yourcommandline> npm install body-parser --save

yourcommandline> npm install mongoose --save

Copy

In your **index.ejs** view file, you should have a form that **sends a user's name and age to the '/users' route**. (Should we use a post or get for this?) Also, we will also be adding the functionality to **get the users from the database every time the index route is hit** so we will need to display users using *ejs.* *Note: we are not giving you this code intentionally, you should be able to figure it out (if you can't then take a look at previous chapters).*

For your reference here is the **server.js** file that we are starting off with. It should look very similar to what you have created in the past.

// Require the Express Module

var express = require('express');

// Create an Express App

var app = express();

// Require body-parser (to receive post data from clients)

var bodyParser = require('body-parser');

// Integrate body-parser with our App

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

// Require path

var path = require('path');

// Setting our Static Folder Directory

app.use(express.static(path.join(\_\_dirname, './static')));

// Setting our Views Folder Directory

app.set('views', path.join(\_\_dirname, './views'));

// Setting our View Engine set to EJS

app.set('view engine', 'ejs');

// Routes

// Root Request

app.get('/', function(req, res) {

// This is where we will retrieve the users from the database and include them in the view page we will be rendering.

res.render('index');

})

// Add User Request

app.post('/users', function(req, res) {

console.log("POST DATA", req.body);

// This is where we would add the user from req.body to the database.

res.redirect('/');

})

// Setting our Server to Listen on Port: 8000

app.listen(8000, function() {

console.log("listening on port 8000");

})

**1. Installing Mongoose**

Now let's get started with Mongoose! Naturally, the first thing that we need to do is *npm install* it and then require it in our **server.js**file. If you used the *package.json* file that we provided you can just run the convenient "*npm install"* and it should install everything for you! If not, you can do it the old school way and run "*npm install mongoose --save*".

**2. Requiring Mongoose**

After we install mongoose, we need to require it. You can add the next block of code anywhere before our routes section. We suggest you add this after we defined the *app* variable.

*var mongoose = require('mongoose');*

Now let's talk a little bit about how our back-end works:

First, we have our **web server** which handles all of the incoming requests and decides which logic units to call upon (in our case our web server and logic units are in the same place -- *server.js*).

Next, we will have our **database server** which is the *"mongod"* that we will have to send requests to (from express). How do we connect the two and how do we make it easy to interact with the database server from express? Mongoose.

**3. Connect Mongoose to MongoDB**

Mongoose has a convenient method: connect!

// This is how we connect to the mongodb database using mongoose -- "basic\_mongoose" is the name of

// our db in mongodb -- this should match the name of the db you are going to use for your project.

mongoose.connect('mongodb://localhost/basic\_mongoose');

Mongoose provides **more structure** to MongoDB by adding **schemas** that we can create that turn into **models for our collections**. **These models specify keys, types, and even validations for documents in a specific collection**. Mongoose also handles appropriate naming (plural for collection names and singular for model names) for us! Neat, huh?

**Note: If you connect to a database that doesn't exist, mongoose WILL create the DB for you!**

**4. Create your Mongoose Schemas**

Let's go ahead and make our first Schema that we will use to model Users. Let's say that each user will have a name that is a string and an age that is a number. The code to create a Schema is pretty simple as below (for now, put this below 'mongoose.connect'):

var UserSchema = new mongoose.Schema({

name: String,

age: Number

})

mongoose.model('User', UserSchema); // We are setting this Schema in our Models as 'User'

var User = mongoose.model('User') // We are retrieving this Schema from our Models, named 'User'

Let's break it down. The **mongoose.model()** method is the most important in this case. Its job is to take a **blueprint object** and, in turn, create the necessary database collection out of the model. **We get this blueprint by making a new schema instance from the mongoose.Schema() object constructor** (see, all that JavaScript paid off!) Notice how the **mongoose.Schema()** method takes a JSON object as its parameter? Well, the structure of that object is how each new document put into the collection will be formatted. You can learn more about the other Schema Types [here](http://mongoosejs.com/docs/schematypes.html). We set the mongoose.model to the *"User"* variable so that we can run model-like methods on it to make all of the CRUD operations easier.

**Note: If you create a model, mongoose WILL create the appropriate collection in your database for you! Even with the appropriate naming (plural for collection names)!**

**5. Use the Mongoose Models to Create / Retrieve / Update / Destroy**

Now that we have our model set up, let's go ahead and use it to save a user to the database via the post to *'/users*'. Let's focus in on the route that we are going to modify:

// This is the route that we already have in our server.js

// When the user presses the submit button on index.ejs it should send a post request to '/users'. In

// this route we should add the user to the database and then redirect to the root route (index view).

app.post('/users', function(req, res) {

console.log("POST DATA", req.body);

// This is where we would add the user from req.body to the database.

res.redirect('/');

})

First, make sure that the *req.body* has a name and age field (we need those two to add the new user to the database). Then let's modify the route to use our User model to save a new user to the database.

**Be aware! You may see the following warning when you save an object using Mongoose:**

DeprecationWarning: Mongoose: mpromise (mongoose's default promise library) is deprecated, plug in your own promise library instead: [http://mongoosejs.com/docs/promises.html](http://mongoosejs.com/docs/promises.html" \t "_blank)

Don't worry if you haven't yet explored the magic of JavaScript promises. They're very cool, but we can get by without them for now. Mongoose, however, is letting us know that its current implementation will break in the future.

Since promises are now native to JavaScript, we can simply overwrite Mongoose's deprecated functionality (you could also use another promise library). How do we do that? See if you can figure it out by traveling to the link provided: [http://mongoosejs.com/docs/promises.html](http://mongoosejs.com/docs/promises.html" \t "_blank)

This line of code might help:

// Use native promises

mongoose.Promise = global.Promise;

app.post('/users', function(req, res) {

console.log("POST DATA", req.body);

// create a new User with the name and age corresponding to those from req.body

var user = new User({name: req.body.name, age: req.body.age});

// Try to save that new user to the database (this is the method that actually inserts into the db) and run a callback function with an error (if any) from the operation.

user.save(function(err) {

// if there is an error console.log that something went wrong!

if(err) {

console.log('something went wrong');

} else { // else console.log that we did well and then redirect to the root route

console.log('successfully added a user!');

res.redirect('/');

}

})

})

There is a lot that is happening in the code above, make sure that you understand everything that is going on. A few key points:

* notice that *"User"* is actually an object constructor (notice the \_"new" *keyword). The*"User"\_, which constructs user objects, have methods that talk to the database!
* *".save"* is the method that actually inserts into the database.
* *".save"* takes a callback function that has an **error parameter so that we know whether or not the insert was successful**(these callback functions really do come up everywhere in JS huh?). This is a pattern that you'll see often -- **any method that interacts with the database will typically have a callback function as a parameter** (the callback function will run when the database operation finishes).

Now test out your code! If it worked then congratulations you successfully filled in the gaps that we intentionally left out! If it didn't work make sure the following things are done:

1. Make sure your *mongodb server* is running (the *'mongod'* command)
2. Make sure your post data matches the data that you are inserting into the database (name and age)
3. Make sure that your form submits a post request to *'/users'*
4. Make sure mongoose is actually installed
5. Check the order of everything related to mongoose (require --> connect --> Schema --> Model --> route)

Great job guys! Now try to modify the root route to render *index.ejs* AND pass it all of the users that are in the database. Then modify *index.ejs* to display all of those users above the form. You guys should be able to figure out how to do this based on your current knowledge of Express, Node, Mongoose, and MongoDB, but we'll give you 1 hint:

// The root route -- we want to get all of the users from the database and then render the index view passing it all of the users

app.get('/', function(req, res) {

User.find({}, function(err, users) {

// This is the method that finds all of the users from the database

// Notice how the first parameter is the options for what to find and the second is the

// callback function that has an error (if any) and all of the users

// Keep in mind that everything you want to do AFTER you get the users from the database must

// happen inside of this callback for it to be synchronous

// Make sure you handle the case when there is an error, as well as the case when there is no error

})

})

You now know how to use Mongoose! Make sure you complete the root route before moving on. The following tabs contain some advanced mongoose methods that you may need for the assignments.