**USEFUL LINKS**

* Express - using a database with mongoose - <https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose>
* Mongoose schema basics - <https://coursework.vschool.io/mongoose-schemas/>
* Mongoose docs, validations - <http://mongoosejs.com/docs/validation.html>
* Mongoose validations and validator - <http://fiznool.com/blog/2014/04/23/mongoose-validations/>
* Mongoose middleware and custom methods - <http://mongoosejs.com/docs/middleware.html>

**IMPORTANT THINGS TO REMEMBER/KEEP IN MIND**

* If you create a model called ‘User’ in mongodb, it will be saved as ‘users’.
* Validations you write in the schema aren’t set off by the update query, meaning that even if the ‘name’ column is required, you could make it blank, and update a record and it would save. Update by default doesn’t execute the validations, though there may be a complicated solution to ensure that it does. However, the .save() method does go through the validations and is the safer option in mongoose.

**CREATING A MODEL - FORMAT**

* **NOTE:** In my full mean apps, the models are each creaeted in their own file within /server/models. I also have a separate file that connects to the db, so you just have to have “var mongoose = require(‘mongoose’);” at the top. You can require other things, such as bcrypt if necessary.
* **BASICS:** You start by requiring mongoose at the top. then you create a variable capitalized and singular, I usually put the name of the model followed by “Schema”. You make this equal to a “new mongoose.Schema()”, and inside the parenthesis, you open an object. Each key in this object will be a column in the model. You can either put just the data type, or as it’s known in mongoose the schemaTypes, without quotes if you want, followed by a comma. Or, if you want to add anything else, including validations, you need to open another object to put the validations and schematype. Once you have put all the columns, you exit the object and put “}, {timestamps: true} );”, putting timestamps true after that will automatically create a createdAt and updatedAt column (the updatedAt is automatically updated with an update query or a .save() method). You can add schema methods if you want, but that’s in a section below. Finally, when you are done, you write “mongoose.model( ‘Final\_name\_of\_model’, SchemaName);”, doing this creates a model in mongoose with the provided columns and info. You then later require these models in the server controller files. See the example below in green for what a model file would look like. Some columns have validations and some don’t to demonstrate the difference:

var mongoose = require('mongoose');

var UserSchema = new mongoose.Schema({

name: {

type: String,

required: true,

minlength: 4,

maxlength: 15,

},

email: String

password: {

type: String,

minlength:8,

maxlength:255,

required:true,

}

}, {timestamps: true});

mongoose.model( 'User', UserSchema);

* **THE DIFFERENT DATATYPES/SCHEMATYPES:** The allowed schemaTypes are: **String, Number, Date, Buffer, Boolean, Mixed, ObjectId, and Array**. Below is an example schema that shows some different schematypes and how you could use them:

var schema = new Schema(

{

name: String,

binary: Buffer,

living: Boolean,

updated: { type: Date, default: Date.now },

age: { type: Number, min: 18, max: 65, required: true },

mixed: Schema.Types.Mixed,

\_someId: Schema.Types.ObjectId,

array: [],

ofString: [String], // You can also have an array of each of the other types too.

nested: { stuff: { type: String, lowercase: true, trim: true } }

})

**BASIC MODELS AND VALIDATIONS**

* **IMPORTANT NOTE:** Validations by default, don’t apply to the update query, there is a complicated way to do it. However, the .save() method in mongoose does activate the validations and will only save if the validations pass.
* **BASICS:** Different schematypes have different built in validators, but all have the “required” validator. Numbers have a min and max validator, while strings have enum (specifies the set of allowed values for a field), match (specifies a regular expression that the string must match), maxlength, and minlength.
* **OVERWRITING ERROR MESSAGE:** You can overwrite the default mongoose error message for when a validation fails, by putting the validation value in an array, the second index will be string and be considered the new error message that mongoose will use if necessary. See an example below for the validations I mentioned in the above bullet and how to replace the default error messages. So eggs, would be a number column, and if you didn’t include it, mongoose would basically send you the error message ‘Too few eggs’. Also, the drink is string and will only accept the values ‘Coffee’, ‘Tea’, and ‘Water’, if a value isn’t given, Coffee will be selected by default.. See below:

var breakfastSchema = new Schema({

eggs: {

type: Number,

min: [6, 'Too few eggs'],

max: 12

required: [true, 'Why no bacon?']

},

drink: {

type: String,

enum: ['Coffee', 'Tea', 'Water',],

default: ‘Coffee’

}

});

* **VALIDATE:** By using the validate property, to perform more complicated validations such as email or password regex. There are two methods to do this, with just a validator function, or with a validator and an error message. the basic and first method has the following format: **validate: function(value){ return regex\_string.test(value) }**, this function returns true or false based on if it matches the regex expression. See an example of the first format below in blue, and then the second format with the error message included in green. Notice, with the second format, in order to include the error message, I pass validate an object instead of just a function. The function is then given to the “validator” key, and the error message to the ‘message’ key. See below:

var UserSchema = new Schema({

email: {

type: String,

validate: function(email) {

return /^[a-zA-Z0-9.!#$%&’\*+\/=?^\_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)\*$/.test(email)

}

}

});

var UserSchema = new Schema({

email: {

type: String,

validate: {

validator: function( value ) {

return /^[a-zA-Z0-9.!#$%&’\*+\/=?^\_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)\*$/.test( value );

},

message: "Email failed validation, you must have....."

}

}

});

**MODEL RELATIONSHIPS**

* **SETTING UP THE MODELS:** Note- this will be for a many to one relationship between models. Whenever you have this type of relationship between models in mongoDB, what you do is have each model store the \_id of the other. Let’s say I have User model and a Comment model, where one user can make many comments. In the User model, I store the comment \_id in an array, with each index being an \_id for a comment, since there are many comments. But in the Comment model, there can only be one user, so I save that User \_id as a single id. We do this in both methods like so: **{type: Schema.Types.ObjectId, ref ‘ModelName’}**, where ModelName is the name of the model you’re trying to reference. In the User model, that would be ‘Comment’, and in the Comment model, it would be ‘User’. That format is the same in both models, the only difference, is for the many Model (User), it is put within brackets [] or inside an array. See the User (Many) model below in blue, and the Comment (One) model in green:

var MessageSchema = new mongoose.Schema({

message: {type: String, required: true, minlength: 10},

name: {type: String, required: true, minlength:4, maxlength: 30},

comments: [{type: Schema.Types.ObjectId, ref: 'Comment'}]

}, {timestamps: true});

var CommentSchema = new mongoose.Schema({

comment: {type: String, required: true, minlength: 10, maxlength: 150},

name: {type: String, required: true, minlength: 4, maxlength: 30},

\_message: {type: Schema.Types.ObjectId, ref: 'Message'}

}, {timestamps: true});

* **IMPORTANT NOTE FOR SETTING UP THE MODELS:** As demonstrated in the examples above, the name of the column/key for the reference is important. In the Many (User) model, you make the name of the column the same as the name of the model it’s referencing but plural, so “Comment” becomes ‘comments:’, and for the One (Comment), the name is the same model it’s referencing but singular and with an underscore in front of it, so “User” becomes “\_user:”
* **USING THE POPULATE METHOD:** This is more for queries and is written in more detail in the doc “mongoose\_and\_mongodb\_query\_basics\_and\_refresher”, but it is important to understand, if I was to query all users with “User.find()”, and a user had created several comments, then the “comments” column for that user would be an array of \_ids, ex: **comments: [582334ecaa2f0e1f1b8cb76d, 58233524aa2f0e1f1b8cb76e, 59c147c5d8c0c018b1194985 ]**, and because of this, we can’t reference or access the comments directly, we would have to loop through the array and query for each one. However, with the .populate() method, mongoose makes this much easier. When used, this method causes the \_ids for the Many model’s array (User in this case), to be replaced with the actual object/instance. So if I did this: **User.find().populate(‘comments’).exec()**, then the array would look like this in the terminal: **comments: [ [Object], [Object], [Object] ]**, you just put the name of the column with the references that you want populated.

**MODEL CUSTOM METHODS**

* **BASICS AND FORMAT:** Each schema can define instance and static methods for its model. Instance methods are the most common, and most are available to every instance of said model. Models are basically just fancy constructor functions, so a method is basically just a prototype method that is inherited by an instance or record of that model. Creating a method is simple, say I have **User** model, then I would write **UserSchema.methods.method\_name = function(){}**. You create all methods after creating your schema, but before you the last line where you actually create the model, ex: “**mongoose.model( ‘User’, UserSchema)**”, so the name that goes before method is the current name of the schema. See the below example in green for how to format a method:

var mongoose = require('mongoose');

var UserSchema = new mongoose.Schema({

name: {

type: String,

required: true,

},

email: {

type: String,

required: true,

},

}, {timestamps: true});

UserSchema.methods.useless\_method = function(){

console.log(this);

return this

}

mongoose.model('User', UserSchema);

* **STILL BASIC FORMAT:** As shown above, you put the methods inbetween the actual schema object, and the line where you make it a model. Also, you have access to “this” within these methods. Each instance has access to these methods, so “this” refers to whichever instance is using the method. For example, if I got a user with “**var user = User.findOne({\_id: …..})**”, then I could call this method like so: **user.useless\_method()** which would call the method, and “this” in the method would refer to the instance that called it.
* **MIDDLEWARE, BASICS AND THE THREE DIFFERENT TYPES:** Middleware, also called pre and post *hooks* are functions that are passed control during execution of asynchronous functions. See this link for the docs - <http://mongoosejs.com/docs/middleware.html>. Basically, middleware are functions that are activated either before (**pre**) or after (**post**) a query or method, and the hook is the actual query/method they happen before or after. There are three types of middleware, each is used for specific queries or methods. The three types are **document, query,** and **model** middleware, **IMPORTANT NOTE:** “this” refers to something different for each type of middleware, keep this in mind. **Document** **middleware** is used for **init, validate, save, and remove**. This type of middleware is basically mean to be used for specific documents or instances, such as when you save or remove a document. As such, the “this” word refers to the document/instance this method was called on. **Query middleware** is used for **count, find, findOne, findOneAndRemove, findOneAndUpdate, update**. This type of middleware is generally used for queries, and the “this” word refers to the actual query in this type of middleware. Lastly, the **model middleware** is only for the **insertMany** model function, and the “this” word refers to the model. **ALL THREE TYPES** of middle ware support pre and post hooks.
* **PRE MIDDLEWARE-** Pre is basically done before the hook or the original query/method. For example, a pre save method will be executed before the document is saved, and determines when it is done. For all pre middleware, the actual method or query( save, find, remove, etc.) won’t be executed until “**done()**” is called. there are two types of pre hooks, serial and parallel. A Serial middleware is executed one after another when each middleware calls next(), though it can also end in a single function. parallel middleware offers more fine-grained flow control, though I have yet to have a need for it. Let’s look at a simple example, let’s say I had a User model with a password, and I wanted to add “-password” at the end of the password column before it was saved. The example method is below, I specify the query or method that activates the middleware, the write the function. You pass something to the variable as a parameter, and it is whenever you call this as a function that the original query or method activates. For example, below I passed “done” to the function, so when I write “done()”, it actually saves the document. If I passed it “next”, then I would need to write “next()”. See my method below in green:

UserSchema.pre('save', function(done){

this.password = this.password + '-password'

done()

})

* **PRE - ERROR HANDLING:** If any pre middleware calls a next or done with a parameter of type “Error”, then the flow is interrupted and the error is passed to the callback. Not sure how or why you would need to to do this, but see the example below in green:

schema.pre('save', **function**(next) {

// You \*\*must\*\* do `new Error()`. `next('something went wrong')` will

// \*\*not\*\* work

**var** err = **new** Error('something went wrong');

next(err);

});

// later...

myDoc.save(**function**(err) {

console.log(err.message) // something went wrong

});

* **POST MIDDLEWARE:** Where a pre middleware executes before the hooked method, a post middleware executes afterwards. This type of middleware doesn’t directly receive flow of control, so not next or done callbacks are passed to it. Post hooks are a way to register traditional event listeners for these methods. I haven’t had any need to use post hooks, so I’ll stop here, but you can see more in the link provided in the middleware basics bullet. See the examples below in red:

schema.post('init', **function**(doc) {

console.log('%s has been initialized from the db', doc.\_id);

});

schema.post('validate', **function**(doc) {

console.log('%s has been validated (but not saved yet)', doc.\_id);

});

schema.post('save', **function**(doc) {

console.log('%s has been saved', doc.\_id);

});

schema.post('remove', **function**(doc) {

console.log('%s has been removed', doc.\_id);

});

**USING PASSWORDS, HASHING, CHECKING, VALIDATION, AND EXAMPLES OF USEFUL SCHEMA METHODS**

* **BASICS:** In order to hash your passwords with bcrypt, you have to require it with **var bcrypt = require (‘bcrypt’)**, in the same file that you create your model. Also, when creating your password column it needs to be a String, required: true, and have a maxlength of 255. Your minlength can be whatever you want. It’s optional to validate it with a regex string, say if you want it to have at least 1 number, uppercase, etc. but not required. Next, we use custom methods to hash and check the password.
* **CUSTOM METHODS TO HASH AND CHECK:** There are many ways to do this, but I prefer to use model methods. I create a method to hash the password and a method to check it for logging in. To generate a hash, the format is as follows with “original\_password” being the variable that represents the password you put into it, you can change the name: **bcrypt.hashSync(original\_password, bcrypt.genSaltSync(8) )**, so in our method, we would return that line, now the next line is for checking the saved hashed password against one that the user inputs, “input\_password” will be the one sent from the view, while “this.password” is the saved one. This line is done within a model method, so “this” refers to the document that is calling the method: **bcrypt.compareSync(input\_password, this.password)**, lastly, to make it cleaner, we create one final pre-save method to hash the password before it is saved. The three methods are shown below in green, these would be in your model file:

UserSchema.methods.generateHash = function(password) {

return bcrypt.hashSync(password, bcrypt.genSaltSync(8));

};

UserSchema.methods.validPassword = function(password) {

return bcrypt.compareSync(password, this.password);

};

UserSchema.pre('save', function(done) {

this.password = this.generateHash(this.password);

done();

});

* **WHAT NEEDS TO HAPPEN IN THE CONTROLLER**: The pre-save method takes care of the initial hashing, so in the controller you just create a new user and save() them. Now to check the password for logging in, so you’re given a password to check from the view and usually a username or email. In the example below I’m using callbacks, just change the format slightly with promises, first you find the user with say: **User.findOne({email: example@yahoo.com}, function(err, user){…})**, one you have the user, you create a variable and make it equal to that user calling the model method you created to check the password, in this case “validPassword”. This method, as shown above in green, takes the password you’re testing, and accesses the password of the document you call it on, and uses the bcrypt.compareSync method, which returns true or false based on if the password matches. So, if you’re user was called “user”, you would do this: **var check = user.validPassword(req.body.password)**, this variable would equal true or false. Then, you simply do an if statement: **if(check) {…} else {…}**, and you’ve got your log in code. You could also put the bcrypt validPassword method straight into the if statement instead of first plugging it into a variable.
* **UPDATING PASSWORD**: A simple, process, but since the update() query doesn’t activate you’re pre-save method, you have to manually hash it. So, for example, if a user wants to change his/her password, first you check that they are the correct user, then you would change the password they sent over like so: **req.body.password = bcrypt.hashSync(req.body.password, bcrypt.genSaltSync(8))**, then you would update said user. This method probably would be different if you used save() instead of update.
* **WHAT THE FILES LOOK LIKE**: Below are simplified version of what the model and controller file would look like with this all thrown in, the model uses a validator and regex for the password, but this is optional. Model is in green, controller is red, comments are in blue:

**SIMPLIFIED MODEL FILE BELOW**

var bcrypt = require('bcrypt');

var mongoose = require('mongoose');

var UserSchema = new mongoose.Schema({

name: {

type: String,

required: true,

},

password: {

type: String,

minlength:8,

maxlength:255,

required:true,

validate: {

validator: function( value ) {

return /^(?=.\*[a-z])(?=.\*[A-Z])(?=.\*\d)(?=.\*[$@$!%\*?&])[A-Za-z\d$@$!%\*?&]{8,32}/.test( value );

},

message: "Password failed validation, you must have at least 1 number, uppercase and special character"

}

}

}, {timestamps: true});

UserSchema.methods.generateHash = function(password) {

return bcrypt.hashSync(password, bcrypt.genSaltSync(8));

};

UserSchema.methods.validPassword = function(password) {

return bcrypt.compareSync(password, this.password);

};

UserSchema.pre('save', function(done) {

this.password = this.generateHash(this.password);

done();

});

mongoose.model('User', UserSchema);

**SIMPLIFIED CONTROLLER FILE BELOW - CREATING AND LOGGING IN**

module.exports = {

create: function(req, res){

var user = new User(req.body);

user.save(function(err)

{

if(err){

res.json(err)

} else{

res.json({message: 'Successfully registered, please log in'})

}

}

)},

login: function(req, res){

User.findOne({email:req.body.email}, function(err, result){

if(err){

res.json({message: false});

} else if(!result){

res.json({message: false});

} else {

var check = result.validPassword(req.body.password)

if(check){

console.log('good login');

res.json({message: true});

} else{

console.log('unsuccessful login');

res.json({message: false});

}

}

})

}

}

**SIMPLIFIED CONTROLLER FILE BELOW - UPDATING**

editPass: function(req, res){ //This is for editing username or password, more critical info

User.findOne({\_id: req.params.id}, function(err, user){

if(!bcrypt.compareSync(req.body.checkPass, user.password)) //check if the password matches

return res.json({message: false, str:"Incorrect Password"})

if(req.body.password) //to encrypt the new password

req.body.password = bcrypt.hashSync(req.body.password, bcrypt.genSaltSync(8))

User.update({\_id: req.params.id}, {$set: req.body}, function(err){

if(err){

res.json({message: false, str:"That username already exists in the database"})

} else {

res.json({message: true, username: req.session.user.username})

} //end of else

}) // end of Update

})// End of initial find One

}, // END OF EDITPASS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*