**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/>

**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. 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

MODEL METHODS AND PRE

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