**USEFUL LINKS**

* Using ES6 Promises instead of callbacks for mongoose queries - <http://erikaybar.name/using-es6-promises-with-mongoosejs-queries/>
* Quora, difference between callback and a promise - <https://www.quora.com/Whats-the-difference-between-a-promise-and-a-callback-in-Javascript>
* software engineering, is there really a fundamental difference between callbacks and promises? - <https://softwareengineering.stackexchange.com/questions/302455/is-there-really-a-fundamental-difference-between-callbacks-and-promises>
* Promises (about) - <https://promisesaplus.com/#point-2>
* Switching out callbacks with promises in mongoose - <https://eddywashere.com/blog/switching-out-callbacks-with-promises-in-mongoose/>
* Avoiding callback hell while using mongoose - <http://www.summa.com/blog/avoiding-callback-hell-while-using-mongoose>
* Using Promises with mongoose ejs - <http://blog.revathskumar.com/2015/07/using-promises-with-mongoosejs.html>
* **stackoverflow, using promises in mongoose routes and creating re-usable methods with queries/promises in them** - <https://stackoverflow.com/questions/38860898/using-promises-in-mongoose-routes>

**ABOUT**

* Callbacks and Promises are two ways to handle asynchronous programming (see bullet below for asynchronous), though both work, promises are generally considered to be better, due to avoiding ‘callback hell’. This concept and the differences between the two is pretty complex and complicated, so this word doc was made to help explain and understand everything.
* Asynchronous programming is when your code stops running synchronously, meaning your code stops running top to bottom. Synchronous is when your code runs from top to bottom, executing each line of code as you go. Asynchronous is when that stops for some reason, or when you have delayed code. An example of this is the http routes you hit in the factories of your full mean project. Ex: “factory.login = function(data, callback){ $http.post('/login', data).then(…..)”, in this example, you send a post to ‘/login’ and the code waits to get it back. When it does, the code after “.then” gets executed. But this is asynchronous code, it stops running normally and waits for the response. Though the code in this example is a callback, either way, if I had a console.log outside the .then() code, it would get run before the code inside of the .then(), hence the nature of asychronouse code
* What is a promise? A promise is a representation of a value that isn’t necessarily known at its creation time. When you use a promise, the function returns a promise rather than an asynchronous call accepting a callback. The code that needs the promise waits until the promise is fulfilled before executing the next step, in Javascript, this is shown as a .then() code.

**WHY PROMISES OVER CALLBACKS**

* Callback hell is when you have nested callbacks, it often gets complicated, hard to read, and very wide before high, meaning it looks like the code in red below:

Department.find({name: 'Development'}, function(error, department){

Manager.find({departmentId: department.\_id}, function(error, manager){

Employee.find({manager: manager.name},function(error, employees){

res.render('employees', {

employees: employees

});

});

});

});

});

* The above code also runs into the issue of, if there is a block or error anywhere in that block, the entire thing gets blocked. Promises not only make your code easier to red, but instead of having an error handler (“if (err)”) for each level, we can have one error handler for multiple blocks. The blue code below is the same as the code above but with promises instead of callbacks:

Department.getDepartment("Development")

.then(function(department){

var Manager = mongoose.model('Manager');

return Manager.findManager(department);

})

.then(function(manager){

var Employee = mongoose.model('Employee');

return Employee.getEmployeesForManager(manager);

})

.then(function(employee){

res.send('/employees', {

employee: employee

});

})

.catch(error) {

console.log(error);

})

.done();

});

* The difference between the pyramid like callback one and the promise block above can already be seen, imagine code where there are even more nested callbacks/promises. Also, to provide an example of not only a mongoose/mongoDb case, but one where the callback needs an error handler for each level but the promise only needs one, see the callback part in red below:

User.findById('123', function(err, user) {

if (err) {

return console.log('error:', err);

}

user.name = 'Robert Paulson';

user.save(function(err) {

if (err) {

return console.log('error:', err);

}

console.log('updated user: ' + user.name);

});

});

* Now see the promise version with only one error handler in blue below:

var promise = User.findById('123').exec();

promise.then(function(user) {

user.name = 'Robert Paulson';

return user.save();

})

.then(function(user) {

console.log('updated user: ' + user.name);

})

.catch(function(err){

console.log('error:', err);

});

* In that example, the .catch serves as the error handler for both .then()s

**BASICS AND FORMAT OF PROMISES WITH MONGOOSE**

* **EXEC:** .exec() is a method that executes the query and returns a promise. You can plug that in a variable, ex: “var promise = User.find({}).exec()”, chain a .then after it, either on the same line or a new one, or return just that result. Queries are executed by either a callback or this method, meaning you can execute it with the above example in quotes, or with “User.find({}, function(err, user){...})”, you could also just put the callback function in the exec(callback here), in which case, it would still be a callback and not a promise.
* **THEN:** technically, queries aren’t full-fledged promises, but they still have a .then() function for yield and async/wait. Basically, a .then() method waits until it’s given what it was “promised”, in other words, after I execute a query with “User.find({}).exec()”, if I have a “.then(function(result){…})” on the next line, the .then() waits until the first query is actually executed and returns a result, and like it’s named, it then does what is inside of it. when writing a .then(), usually you put a function, ex: “.then(function (user){..}), it doesn’t matter what you put as the function parameter, I put “user” in this example, but it represents or is actually the result of the .exec() or .then() method ahead of it. IMPORTANT: unlike .exec(), .then() doesn’t return anything by default, so you need to return something at the end of it, or the next thing, whether it be .then, .catch, or something else won’t have anything to work with.
* **CATCH:** this is used to catch errors, basically the “if (err)” part in callback functions. There is little documentation on it, though from the examples I’ve seen, it can be used for multiple .exec or .then methods. format is typically used after a “.then()”, and is “.catch(function(err){….do something….})”
* Basically, you exec a query with the model by using “.exec()”, example with a User model: “User.find({}).exec()” , you then chain off of it with a .then() method, which you can keep chaining with a .then()s. Make sure you use return at the end of each .then(). At the end, you can use a .catch() to catch any errors though there are other ways you could do it. ex:

User.findOne({….}).exec()

.then(function(user){

do something…..

return [user, something\_else]

})

.then(function(result\_from\_first\_.then){

do something….

return something

})

.catch(function(err){

do something with error…

})

return ….

* That’s the basic format, you could return whatever you want, probably the result from all the queries, the link listed above: using promises with mongoose: <http://blog.revathskumar.com/2015/07/using-promises-with-mongoosejs.html>, is a good example

**USING PROMISES IN MONGOOSE/ SWITCHING OUT YOUR CALLBACKS WITH PROMISES**

* **GREAT EXAMPLES:** also listed above: stackoverflow, using promises in mongoose routes - <https://stackoverflow.com/questions/38860898/using-promises-in-mongoose-routes>, switching out callbacks with promises in mongoose - <https://eddywashere.com/blog/switching-out-callbacks-with-promises-in-mongoose/>, using promises with mongoose ejs - <http://blog.revathskumar.com/2015/07/using-promises-with-mongoosejs.html>, stackoverflow, best practice of using promises with mongoose - <https://stackoverflow.com/questions/38042964/best-practice-of-using-promise-with-mongoose>
* **IMPORTANT NOTE:** In the examples below, the callback code will be in red, the promise code will be in blue, and comments in green
* **EXAMPLE ONE**: In this example, which I take from the switching out callbacks with promises in mongoose link above, I first find a user, update the name, and save it.

// error first callback style

User.findById('123', function(err, user) {

if (err) {

return console.log('error:', err);

}

user.name = 'Robert Paulson';

user.save(function(err) {

// yet another err object to deal with

if (err) {

return console.log('error:', err);

}

console.log('updated user: ' + user.name);

// do something with updated user

});

});

* In the above example, I have two error handlers (two if’s), and the entire block of code is contained within the parentheses opened by “findById”, which makes it hard to tell by looking what ends where. And this is with only one level of nesting, the code below is the same thing but with promises. It also uses model queries that return a promise via the .exec() function:

var promise = User.findById('123').exec();

promise.then(function(user) {

user.name = 'Robert Paulson';

return user.save(); // returns a promise

})

.then(function(user) {

console.log('updated user: ' + user.name);

// do something with updated user

})

.catch(function(err){

// just need one of these

console.log('error:', err);

});

NOTE: the first line “var promise =” doesn’t have to be plugged into a variable, it could also be “User.findById(‘123’).exec()” and then, on the next line, by starting with “.then()”, you chain off of the .exec(), also “.exec()” by default returns a promise. Putting it in a variable, then using the “.then()” off of that variable isn’t necessary, in most examples I see, they don’t use a variable.

* Notice how there is just one error handler for both promises, the .catch(), promises are often more readable, especially when you have even more nested queries. Also notice, in the first promise, you return something, always return something at the end of a .then()
* **EXAMPLE TWO:** this code is from using promises with mongoose - <http://blog.revathskumar.com/2015/07/using-promises-with-mongoosejs.html>. In this example, I fetch a User, then a Project with the user\_id, and an Issue with that project, so I work with 3 models. Also, this isn’t full mean, due to rendering the page directly at the bottom, but the code is still relatable. See the callback version below:

var User = require('./models/user');

var Project = require('./models/project');

var Issue = require('./models/issue'); //these are just calling the models from other files

exports.index = function (req, res) {

var username = req.params.username;

var project = req.params.project;

User.findOne({username: username}, function(err, user){

if(err) { //first error handler

console.log(err);

return

}

Project.findOne({name: project, user: user.\_id}, function(err, project){

if(err) { //second error handler

console.log(err);

return

}

Issues.find({project\_id: project.\_id}, function(err, issues){

if(err) { //third error handler

console.log(err);

return

}

res.render('./views/issues/index', {user: user, project: poject, issues: issues});

//This renders the view directly, we would do this differently in full mean

})

});

});

}

* In the example above, you can start to see how nested queries can get messy, but there isn’t a lot done inside the actual queries themselves. Also, notice how we always have access to the result of each query after said query, so we don’t have to keep track of them with the callback method, since each is inside the scope of the initial query, and each query resides in the scope of the one before. Now, see the promise version below:

var User = require('./models/user');

var Project = require('./models/project');

var Issue = require('./models/issue');

exports.index = function (req, res) {

var username = req.params.username;

var project = req.params.project; //these two lines aren’t necessary, just cleans up the code a bit

User.findOne({username: username}).exec()

.then(function(user){ // “user” is what the first query returned or gave us

return Project.findOne({name: project, user\_id: user.\_id}).exec() //we could actually plug this into another variable so we wouldn’t need the .then() statement below

.then(function(project){ //this is actually a second .then still in the scope of the first one, so it stil has access to “user”

return [user, project]; //now we’re passing the results from the first and second query to the next .then()

});

})

.then(function(result){ // “result” here is equal to the “return[user, project]” above

var project = result[1]; //this is equal to the result from the Project.findOne query above

return Issues.find({project\_id: project.\_id}).exec()

.then(function(issues) {

result.push(issues); // now result would look like: “[user, project, issues]”

return result; //we return result as it looks in the comment above

})

})

.then(function(result){ //this .then just takes the result from our 3 queries and plugs them into variables, would could skip the .then altogether by putting the res.render below in the previous .then() or even trying to put the results from the last then(0 into a variable, though I’m not sure how that could be done.

var user = result[0];

var project = result[1];

var issues = result[2];

res.render('./views/issues/index', {user: user, project: project, issues: issues});

})

.then(undefined, function(err){ // in theory, this can be replaced with a .catch statement

//Handle error

})

}

* The above code could be changed slightly, such as changing the last .then into a .catch, we could also incorporate the .populate method. This code isn’t perfect, but it serves as a good example. also, notice how I have to keep track of the different query results in this one, most likely because each .then is within its own scope, see the comments I put to track the variables.
* **EXAMPLE THREE**: This code is taken from the blog, avoiding callback hell while using mongoose - <http://www.summa.com/blog/avoiding-callback-hell-while-using-mongoose>. In this example, we have a student and a course and we want to enroll the student into the course. But before enrolling the student, we need to make sure that there are enough seats available in the course. See the callback version below:

var Student = require('./student.model');

var Course = rqurie('./course.model');

//Let's assume we are using Express framework for nodeJs

function enrollStudent(req, res, next) {

//First we need to make sure that student exists and load the user.

Student.findById(req.body.studentId, function(err, student){

if(err) return next(err);

//Now we need to load the course

Course.findById(req.body.courseId, function(err, course){

if(err) return next(err);

//Next we need to check if there are available seats in the course

if(course.isSeatAvailable()){

//Enroll student into the course and save the course

course.enrolledStudents.push(student.id);

course.save(function(err){

if(err) return next(err);

return res.json({message : 'Enrollment successful'});

});

} else {

//Call error handler

return next({message : 'No seats available'});

}

});

});

}

* You can see the indentation getting worse, and the effect is lessened in a Word doc. We also have the usual multiple error handlers. the more nested things get, the more unreadable callbacks become. Now see the promise version below:

function enrollStudent(req, res, next) {

var student;

//Load the user

Student.findById(req.body.studentId).exec() //notice how we don’t have to return this or put it in a variable, the “.then()” automatically chains off of it.

//Capture student and load the course

.then(function(studentFromDb){ // “studentFromDb” is the result of the first findById query.

student = studentFromDb

return Course.findById(req.body.courseId).exec(); //we don’t return studentFromDb, so if we didn’t plug it into the student variable, we’d lose it

})

//Check if there are available seats in the course

.then(function(course){ //this is equal to the course we found in the last query

if(course.isSeatAvailable()){

//Enroll student into the course

course.enrolledStudents.push(student.id);

return course; // if it worked, we return the course so the next .then() has access to it.

} else {

//throw an error

throw new Error('No seats available');

}

})

//Save the course

.then(function(course){ //the same “course” that the last .then() used.

return course.save(); // .save returns a promise just like .exec() does, but if we didn’t put the return in front of it, we could have kept going in this .then(), he most likely went to another .then() so if there was an error, the .catch would get it.

})

//Send the response back

.then(function(course){

return res.json({message : 'Enrollment successful'})

});

//Catch all errors and call the error handler;

.then(null, next);

} //this could be replaced by a .catch()

* In the example above, pay careful attention to what gets returned from each .then(), and how it affects the next .then() or rather what it gives it. Using promises makes the code easier to read and keeps it from indenting too much.

END