You are writing a backend service for your web app and you need to fetch data from your mongo cluster. The problem is, you do not want your request to be returned in a synchronous manor — instead I want mongo queries to be carried out, get the returned result, handle it and return the JSON to my front end when it is ready.

You want to conform to ES6/7 javascript standards to maintain your app code and stay relevant; you need to implement promises, async and awaitfunctionality, wrapped around your db.collection mongo requests.

**Breaking down the process**

Before the full example and explanation of a fully-fledged asynchronous API request, I want to briefly revisit promises and using async/await in javascript.

Promise

Promises give us a way to handle asynchronous processing in a more synchronous fashion. They represent a value that we can handle at some point in the future; it will eventually be returned, or *resolved*.

If we call a promise and console.log it, we will be greeted with a *pending promise*. The promise has not yet been *resolved*. It is in the middle of completing what we code it to do. When it has resolved we will be able to retreive the data we originally intended the promise to return to us.

Promises are immutable, the handler cannot be changed. We are also guaranteed to receive a return value: either what we intended or an error.

*We will write our promises inside ES6 functions, and then asynchronously call it using await and async.*

What does a promise look like? Something like this:

var myPromise = () => (  
 new Promise((resolve, reject) => {  
   
 //do something, fetch something....  
 //you guessed it, mongo queries go here.  
 db.collection('your\_collection').find(...)

//I can continue to process my result inside my promise  
 .then(function(result){

//another query can be called based on my result...  
 return updatedResult;

})

//This promise may take a while...  
 .then(function(result){

//post processing, non related mongo code...

//when you are ready, you can resolve the promise.  
 resolve(result);  
 });  
 })  
);

Notice the second handler (*typically named* reject). It is a function to call to reject the promise if it can't resolve the future value.

We could expand the previous psuedocode to account for rejecting unwanted data:

//when you are ready you can resolve the promise.

var somethingWentWrong = (dataReturned == null);

(somethingWentWrong)  
 ? reject('something messed up')   
 : resolve(result);

Now let’s move onto asynchronously processing our promises.

async / await

As you can see, the async and await keywords are absent from our Promise code. We use them to configure asynchronous functions that call our promise and wait for it to complete, like this:

var callMyPromise = async () => {  
 var result = await (myPromise());  
 return result;  
};

See how simple that was? Some articles online make the process look rather complicated. It is not — separate your promise declarations and your asynchronous functions. Make things simple to read and build upon; your team will appreciate it.

So the last piece of the puzzle is to coherently put everything together so we can finally return our API request, which looks something like this:

callMyPromise().then(function(result) {

//close mongo client  
 client.close();

//feel free to process your final result before sending  
 //it back to your front end

//return the API request  
 res.json(result);  
});

**Putting everything together**

Let’s put everything we just went through together to create a full API request. Let’s say I am using Express as my backend service:

router.post('/api/get\_data', (req, res, next) => {

try {

MongoClient.connect(connectionStr, mongoOptions, function(err, client) {

assert.equal(null, err);  
 const db = client.db('db');  
   
 //Step 1: declare promise  
   
 var myPromise = () => {  
 return new Promise((resolve, reject) => {  
   
 db  
 .collection('your\_collection')  
 .find({id: 123})  
 .limit(1)  
 .toArray(function(err, data) {  
 err   
 ? reject(err)   
 : resolve(data[0]);  
 });  
 });  
 };

//Step 2: async promise handler

var callMyPromise = async () => {  
   
 var result = await (myPromise());

**//anything here is executed after result is resolved**  
 return result;  
 };

//Step 3: make the call

callMyPromise().then(function(result) {

client.close();  
 res.json(result);  
 });

}); //end mongo client

} catch (e) {  
 next(e)  
 }

});

module.exports = router;

Some points about this example:

* The entire process is wrapped in a try catch so I can handle any errors that occur.
* res.json returns the result of my data as a JSON object.

Now, what we have done here*is mix async and await features with ourthen() callback functions.****However we could choose to utilise only one of these.***

So why did we use both in the example above? Because it demonstrated how we can await an async function to resolve, **which are also treated as promises**.

Let’s explore how we can optimise the example below.

**Using promises without async / await**

Now, we could in fact remove the async / await keywords here along with step 2, and simply continue with a then() block after the promise is called:

//Step 1: declare promise

var myPromise = () => {  
 ...  
};

//omitting step 2

//step 3: make the call

myPromise().then(res => {  
 client.close();  
 res.json(result);  
};

Indeed, this is cleaner syntax. In reality, your promise will be imported from an external modules file, therefore step 3 will be the only code present at your routes level.

**Using async / await only**

So why would we use the async / await keywords in this example?

Check out the rewritten example below.

We declare our promise as step 1 like before, but we then utilise await to pause execution until myPromise is resolved, before closing the mongo client and resolving the API call.

Notice the async keyword is now being used in the router callback function on the first line:

router.post('/api/get\_data', **async** (req, res, next) => {

try {

MongoClient.connect(connectionStr, mongoOptions, function(err, client) {

assert.equal(null, err);  
 const db = client.db('db');  
   
 //Step 1: declare promise  
   
 var myPromise = () => {  
 return new Promise((resolve, reject) => {  
   
 db  
 .collection('your\_collection')  
 .find({id: 123})  
 .limit(1)  
 .toArray(function(err, data) {  
 err   
 ? reject(err)   
 : resolve(data[0]);  
 });  
 });  
 };

//await myPromise  
 var result = await myPromise();

//continue execution  
 client.close();  
 res.json(result);

}); //end mongo client

} catch (e) {  
 next(e)  
}

});

module.exports = router;

Which style do you prefer? then() may appear more readable for some, whereas await may look cleaner more minimal code for the more experienced programmer.

**Ready to Continue?**

I continue exploring the promises, async and await concepts in follow-on articles, that expands on the concepts in this article, to creating a library of promise based exports for your API calls.