## **TESTING DATABASE OPERATIONS**

### **Introduction**

When the code that you are testing has to touch a database, the amount of setup that you have to do gets quite a bit more complicated. Obviously you don’t want to run your testing code on your production database because of the risk of compromising your user’s data. In this lesson we’re going to set up a new in-memory version of a mongo database and then tell our app to use that when running our tests.

### **Learning Outcomes**

By the end of this lesson, you should be able to do the following:

* Explain the purpose of using a separate database for testing.
* Use mongodb-memory-server to create a testing database
* Familiarize yourself with an alternative method of database setup for your testing environment.

### **But do you even need to test that?**

Before diving in, you might want to consider whether or not the database operations you’re testing even need to be tested in the first place. If you are simply reading and writing straight from a database using mongoose or some other db module, you might not really need to test that code. Mongoose (and presumably all other popular db modules) already has [plenty of tests](https://github.com/Automattic/mongoose/tree/master/test) for all of its actions, so if you are just serving up a JSON API and all you’re doing is leveraging functions from another module then those operations are already covered.

If your queries are complicated, you might justify adding tests to make sure that you are using them correctly and that the code you have written is doing what you intend it to, and if you are using any of your own code to do some filtering, sorting, or other manipulations of the data you will want to test that as well. In the case of your own code, however, it would be better if you could pull those things out into their own modules, separate from your database operations so you can test them without messing with the database.

### **mongodb-memory-server**

There are cases, however, when you are going to want to test things that touch your database, so this lesson is going to cover how to do that.

We’re going to use an npm package called mongodb-memory-server. You can see the specifics on their [github repo](https://github.com/nodkz/mongodb-memory-server), but basically this package will spin up a fresh in-memory mongoDB server that you can connect to with mongoose, and then use for your testing environment. Since it’s creating a fresh DB every time you don’t have to worry about trying to keep your data in sync, or polluting your production database.

Setting it up is actually pretty simple, but there are a few things you need to do. First, in your actual app, you need to move your mongo/mongoose setup into its own file as seen in the simple example below.

//// mongoConfig.js const mongoose = require("mongoose"); const mongoDb = `YOUR MONGO URL`; mongoose.connect(mongoDb, { useNewUrlParser: true }); const db = mongoose.connection; db.on("error", console.error.bind(console, "mongo connection error"));

The above code should look very familiar to you by now.. it’s the same setup code we’ve been using all along. The only difference is that we’ve moved it out to its own file so that in our test files we can use a *different* config file that sets up mongodb-memory-server for us. All you have to do now is require("./mongoConfig") in your app.js file.

Next we need to create a separate config for our testing environment. The config file that you can find on the mongodb-memory-server repo README should work just fine. Below is a slightly edited version of it. Copy this to a new file called mongoConfigTesting.js

//// mongoConfigTesting.js const mongoose = require("mongoose"); const { MongoMemoryServer } = require("mongodb-memory-server"); const mongoServer = new MongoMemoryServer(); mongoose.Promise = Promise; mongoServer.getConnectionString().then(mongoUri => { const mongooseOpts = { // options for mongoose 4.11.3 and above autoReconnect: true, reconnectTries: Number.MAX\_VALUE, reconnectInterval: 1000, useNewUrlParser: true }; mongoose.connect(mongoUri, mongooseOpts); mongoose.connection.on("error", e => { if (e.message.code === "ETIMEDOUT") { console.log(e); mongoose.connect(mongoUri, mongooseOpts); } console.log(e); }); mongoose.connection.once("open", () => { console.log(`MongoDB successfully connected to ${mongoUri}`); }); });

Now, if your tests are set up similarly to the tests in our last project you can simply require this file at the top of your testing file, then, when testing any operations that work on your mongo database they will use this testing one instead of your real one.

### **A couple of notes**

Since you are starting your tests with a fresh database it will probably be useful for you to use a beforeAll function in your testing suite to add a couple of items to the database before running tests.

This is also not the only way to set up a testing environment! If you are using nconf, or command-line arguments or anything else to set up your development and production environments, you could easily add a testing environment that uses this mongodb-memory-server. The [Jest Docs](https://jestjs.io/docs/en/mongodb) demonstrate an alternative (but similar) setup to the simple one we have here. The common element here is that no matter how you accomplish it, our goal is to use this alternative DB when running our tests.