14)Project overview

Go to users folder and run this-

**npm init**

it will create package.json file. we will asked couple of question, use default. Later we will come back to it and update some things.

Now run this-

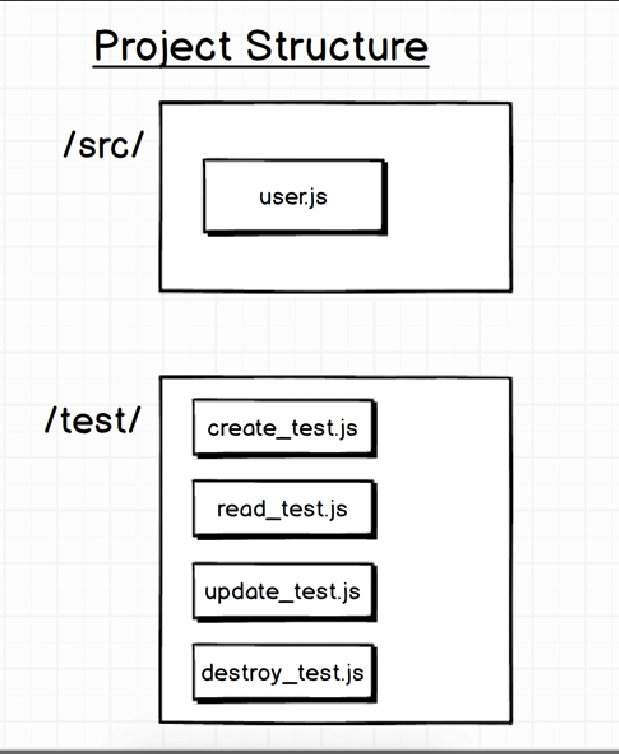
**npm install mocha nodemon mongoose --save**

lets talk about our application here-

our goal is to have mastery of CRUDS operations in mongo database. In our application we will not have any frontend. We will create a mongo database with collection of users. Then we will wire up this collection of users with mongoose library. Once we wire them we will use mocha testing framework to test our entire setup and make sure that we actually inserting records in our database.

Lets talk about our project structure-

User.js will contain bunch of code that will map mongo and mongoose togather. After that we will start writing test files in separate folder called test. We will have 4 files there. One for each type of crud operation. See structure-



15)Test helper File

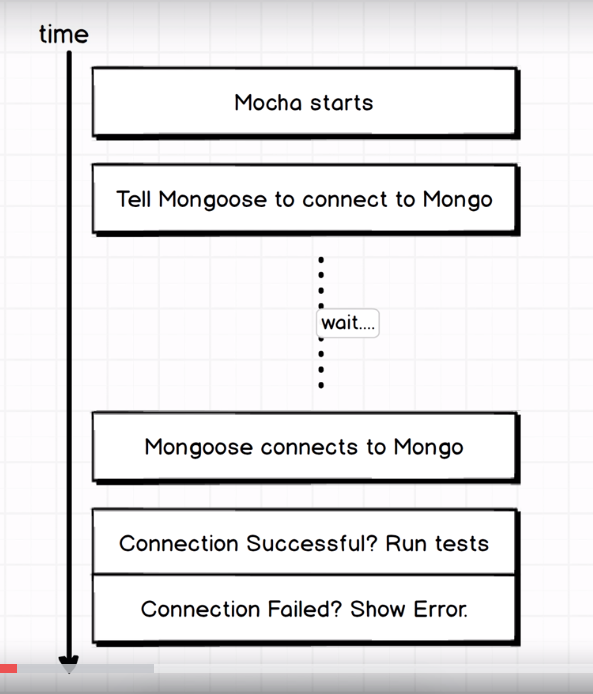
Mocha framework is most popular for doing any kind of java script testing around nodejs.

We have also installed nodemon. It will help us with our test. We will come back at it.

Now create folder src and test. Inside test we are going to do test envirement setup. So create this file - test\_helper.js. inside this file it is customary to place any code that we might need to have to set up our testing environment. It is customary to name it like that.

Here we will do some initial setup for test.

Lets see how our code is going to execute when we run a command to test our project.



When we run a command to test our project, all our code loads and mocha library will start up. It says ok, it’s my show now, I will test this project. But before that happens we have couple of steps that we have do first. First we need to connect mongoose with mongo db and it will take some arbitrary time. But the point here is that inside our test helper file we are going to do a little bit of setup code to tell mongoose to connect to mongo. Then we see if connection is successful, of yes then we will run some of tests. If connections fails then we will show some error. So that’s our goal. This is the setup that we are going to do in test helper file.

Test\_helper.js-

const mongoose = require('mongoose');

mongoose.connect(

"mongodb://localhost:27017/users\_test", {useNewUrlParser: true }

);

mongoose.connection

.once('open', () => console.log('Good to Go'))

.on('error', (error) => {

console.warn('warning', error);

});

16)Mongoose connection helper

Here we will explain the code that we had in last lecture.

In diagram first step was to connect with mongodb. With connect function we do that.

Here local host means we want to connect with mongo db running on our machine. If we want to connect with mongo erver on some other ip, instead of localost we will give ip and port of hat machine ex- 12.45.67.89:345667

Now after local host we have users\_test. This is name of our database remember we can multiple database in single instance.

You dnt have to create a database ahead of time. you can just start inserting data. Database will be created if it is not there.

Now we have sent request for connection. Now it might take some time for connection. Database can be on other side of planet. This represents the wait phase in diagram.

Now to get message when conncetion has succed or failed, we use once and on event handlers. Once means watch for mongoose to emit a event called Open one time and once it does, then run this callback. Same is for error. Now we are not using open and error events on our own.they are very particular events. They are part of mongoose library.

Last thing we want to talk about is fat arrow function. they are identical to normal functions.

With mongoose there are couple of occasions when you do not want to use fat arrow function. I will point them out.

Now go to file path and run it by-

**node test\_helper.js**

if you db is running and mongoose is able to connect , you will see this-

good to go

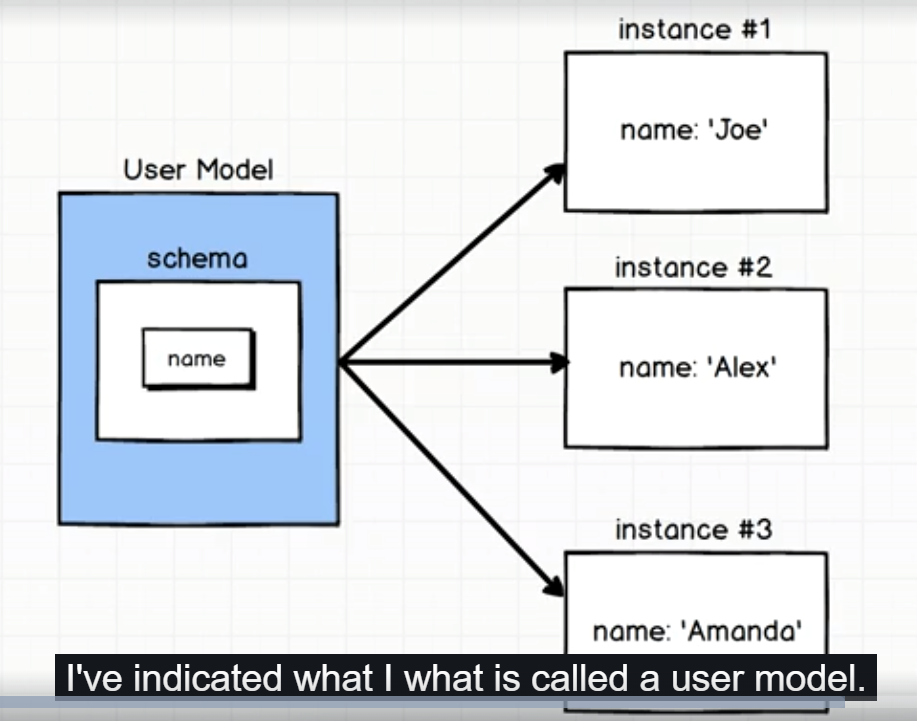
Both once and all are event hanlers. Once means watch for mongoose to emit an event called open, one time, once it is emitted run this function that we have passed.

We also say watch for event called error and then execute that function. both open and error are event that are defined in mongoose library.

17)Mongoose Models

We used mongoose to connect our db. now we will use mongoose to create new collection of data inside our db. we will create users collection.

Before creating a new collection of users , we need to understand our mongoose works.



On left hand side we have user model. We use mongoose to create models that represents a record in a very specific collection. The user model here represents all of the records that sit inside of user collections. As of know we dnt have user collection in our db but we will create it shortly.

We can think of userModel representing entire collections os users because it has bunch of functions(on model that we will create through mongoose), for working with all records in user collections. Usermodel will also be used to create objects that represent single instances or single records within our collections as well. On right hand we have indicated that usermodel will be used to create single instances in data base. So these are new users.

So we have UserClass or UserModel and then we have user instances which represent single records in our collection.other thing to remember about models is that , they have very important property called schema. The schema tells us exactly what properties we expect each record in collection to have and what type of data we expect it to be. In diagram, for my schema on user model , I am indicating that every user should have a name. so we also need to fpus on how to create this schema.

In src folder , create a new file – user.js. here we will define user model.

User.js-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

This Schema will allow us to create s=Schema for user model.

18)More on Models

Remember that user model represent all of data in a singe collection in our mongo database. When we will create user model, mongoose will automatically create a user collection in data base. single instance of user model represent the single user data(document in mongo db).

Lets look into our user.js file. here we will first create schema for our user. Here we will just have name property but with time our schema will become complex. here we say that I expect every user in my collection to have name property and I expect it to be string.

Code-

user.js-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: String

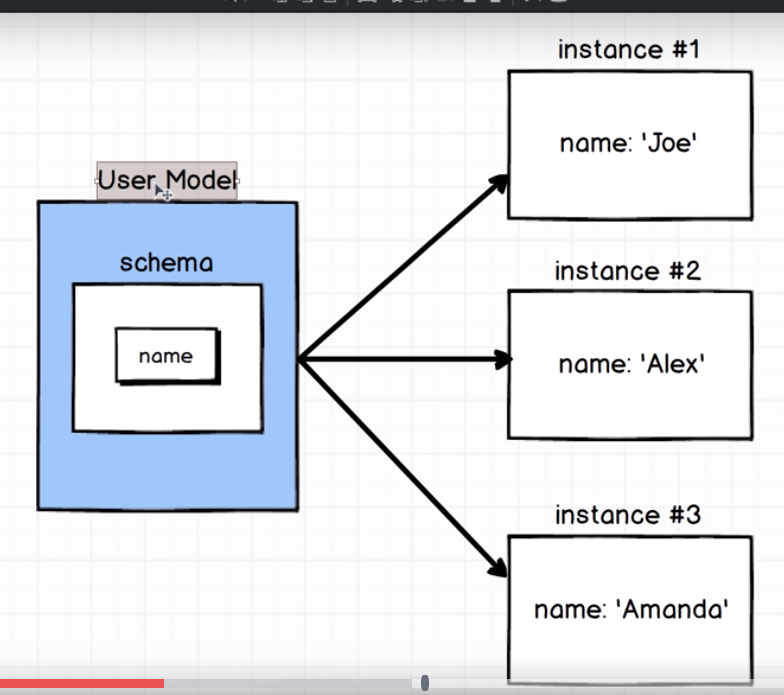
});

const User = mongoose.model('user', UserSchema);

module.export = User;

String here is refrence to base string class or object(I should say) inside js.so this is a special global variable that you have access to inside of language. All js implementations have access to this , it is not specific to mongo or mongoose. Now we have schema now we can feed it not mongoose to create user model. Schema is small portion of what makes up a user model.

Taking look at this diagram, to make sure it is nice and clear-



UserModel represents entire collection of data that’s inside of our database , schema is very small component of that model. The schema is only something in here that kind of tells the model hey here’s the different properties I expect this thing to have. So schema is only a small portion of what makes up an overall usermodel.

Then we create model. We name this thing with string user and then I pass user schema to let this thing know that we expect it to follow our schema.

const UserModel = mongoose.model('user', UserSchema);

this is where real magic starts happening. mongoose starts doing lot of work behind the scenes. Mongoose says, mongo do you a collections named user? If not then go ahead and make it. The string that we pass to model as first argument controls what collection is called inside mongoose. Remember mongoose is just wrapper round mongo database , to make things easier. So right now it is making things easier for us. So we dnt have to go to mongo db and create collection, mongoose is doing that for us.

Second argument instructs mongoose about schema that we expect this collections of users to follow. Finally we return the result of model function to User variable. We will refer to this User variable as User Class along with User model. We will use this terms interchangeably user class and user model. it does not represents any particular user inside our application. It represents the entire collection of data that’s sitting inside our database.

Now I want to make sure any file in my application can get access to User Model that I have created here. Now if any file do this-

require(‘user.js’)

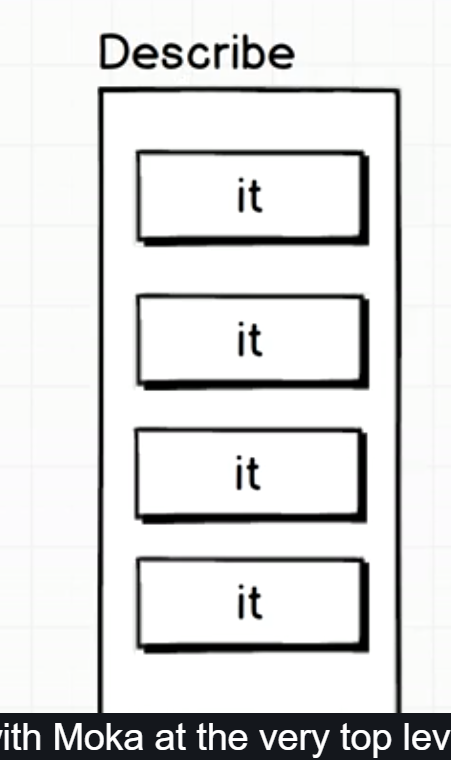
then it will access to User. If any file wants to create,find ,update user, it can do so with User Model.

19)basics of mocha

Now we have created a model, I want to write little code inside of my test directory to test this thing and make sure that we can insert a record in our database see that it exists . to write this test we need to learn little more about how mocha works.

Create a new file, create\_test.js inside test folder. Purpose of this file is to write some test that are going to make sure that we can create a new user and save it into our database.

Before that we need to learn how moka expects us to structure our tests.



This is diagram of type of structure that we are going to write for all of our tests when we start writing code with mocha. At very top of every test file, we are going to have, what we refer to as describe block. Block is little misleading term. It would be better be labelled as describe function. so we are going to pass describe, we are going to call function called describe and we are going to pass it a function. inside that function we will have variety of it blocks. It block like describe would be better referred to as it function. so we will call function called it and inside of this function we will put in little bit of syntax or little bit of code to test a very specific part of the code that we have inside of our user model.

Both describe and it function are provided automatically by moka. So we dnt need to require anything. Lets write our test-

describe('Creating records', () => {

it('saves a user', () => {

});

});

Describe takes 2 argument- one is string (that describe test that we are about to write) and second is function.

This test is for creating user , so we pass this string. The purpose of this string here is just to modify the testing feedback that we get from moka. This is just to see which test are breaking when we see our test report. It has nothing to do with how our test are written, ran or anything like that. It is solely for the purpose of uus figuring out what tests are breaking when we look at our test report . so give sensible names so that you can figure out where things are going wrong.

So inside our function that we passed to describe, we will place many it blocks, remember it is just a function like describe. It takes same argument like describe.

It block is absolute key to all testing inside moka. When moka sees a it block, it says- ah, user or developer is trying to run some kind of test inside the function that we pass to it.

It will queue up all of different it functions inside the entire test suite and will run them, one at a time.

Inside of every it function that we have made, we need to make what is called an assertion. An assertion is a piece of code where we say I really hope, this value is equal to this value. The idea behind the assertion is to actually do the test and actual part that says like hey is my code running appropriately.

By default moka does not include any tooling inside of it, for automatically giving you the ability to make an assertion and I say that because we automatically got access to describe and it. But same thing is not true for making an assertion inside of it function. in next lecture we will see how we make assertion inside a text file.

20)Running Mocha Tests

In last lecture we made a file named create\_test.js. it is meant to create test to makw sure that our user model can correctly create a new record in database. This explains the name.

So we said in last lecture that inside it we need to make an actual assertion something that is going to compare one value to another and then moka is going to interpret the result of that comparison and tells us either our test failed or passed. The last thing that I mentioned was moka gives us global access to describe and it function but it does’nt gives us anything to actually make an assertion. So to make assertion w ehave to import assertion library at top of our file. this was installed , when we installed moka.

The assertion is going to be expected to be passed a bollean of some type. So traditionally we will compare 2 values and say I assert or I expected that like this value is equal to this value. if its true then test is passed, if it fails then test is going to fail. Code-

const assert = require('assert');

describe('Creating records', () => {

it('saves a user', () => {

assert(1+1 ===2);

});

});

To run some test when put some assertion togather, we use cmd to run our test suite.

Run this command-

**npm run tests**

we will get error.by default we usually use the convention of npm run test to run our entire test suite inside of our project. But at this point we have’nt done any setup for the command **npm run test** yet. So we need to do this. We do this in package.json.in package.json we have script section –

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

This is from where our error message came from. Only existing script that we have now is the test script and when it ran , it prints out error- “no tests specied” and exits the process with an error code. Replace it with –

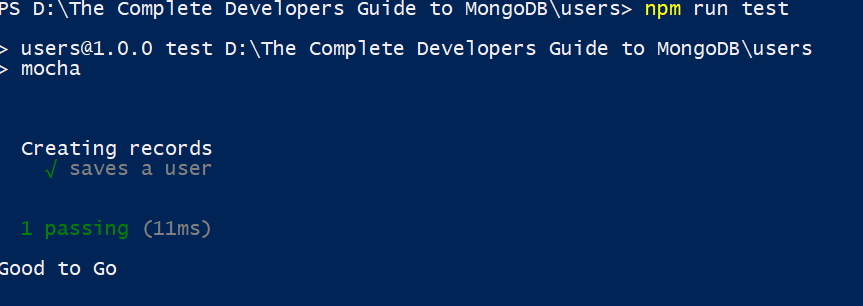
"scripts": {

"test": "mocha"

}

Now go t users folder and run-

Npm run test. Output-



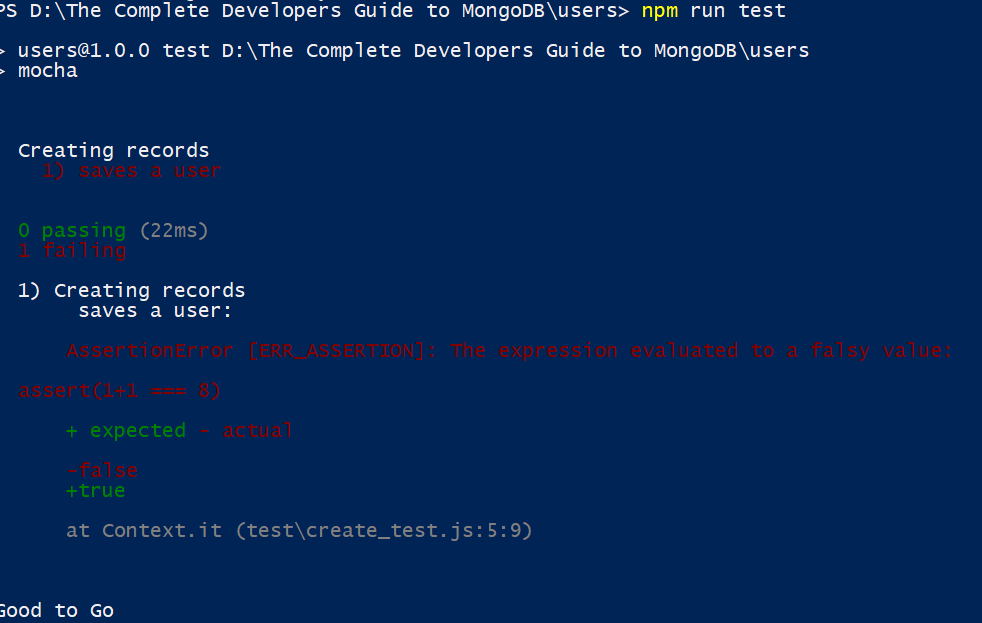
“saves a user” is string that we passed to it function. now lets fail this script chnge assertion and re run test.

it('saves a user', () => {

assert(1+1 === 8);

});

Output-

this is why we pass strings to it and describe. It helps us understand where our scripts are failing.

21)Creating model instances

In last lecture we created a test file and executed a couple of tests inside it. In this lecture we will write test around creating a user and saving it to database. So remove assert statement from it block. Now inside of our test file we want to have access to all the code that is written inside of user.js. right, like the whole purpose of this test is to make sure that we can use this user model that we created(in user.js) to save something in database. So it makes sense to have access to code of user.js inside this file. so we use require sattaemnt to access User model. Now we can use User Model to create a new user inside of it model. We will create user then we will save it to database then we will use some amount of code to sy , hey did this thing actually get saved? So these are 3 steps. First we created a user. This is simplest way to create a user. There are fancy ways to created users, that we will see later. There are lot variations. Now we have our user. Now we need to save it in database. We will see it in next lecture. Code-

const assert = require('assert');

const User = require('../src/user');

describe('Creating records', () => {

it('saves a user', () => {

const joe = new User({

name: 'joe'

});

});

});

22)Saving users to mongo

Inserting records into our database, I will say that it is challenging and simple at the same time. it is simple because basic syntax for saving a record in db is really easy to remember but it is a little more complicated because there is a lot of consideration around saving a model that are going to come into play as we start to explore some of the advanced functionalities of mongoose. For now its very basic thing to save model.

We called save function on joe. That means joe is not simple object, it will have bunch of functionality tied to it by default. It has ton of functions attached to it that we can call at any given time. one of these methods is save. Lets run this code. Here goal is we just want to check that record is inserted in db. here we are not running test case. It is diversion. Code-

const assert = require("assert");

const User = require("../src/user");

describe("Creating records", () => {

it("saves a user", () => {

const joe = new User({

name: "joe"

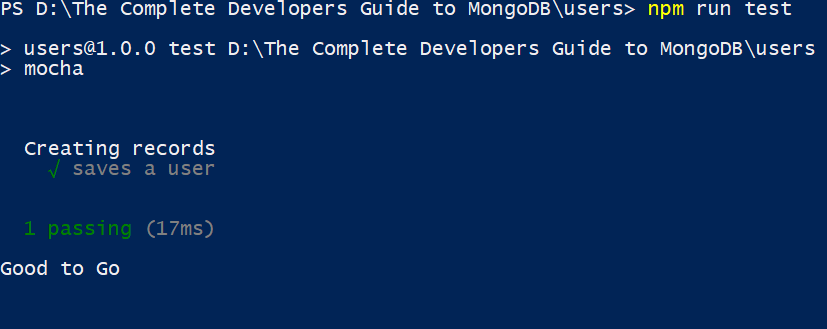
});

joe.save();

});

});

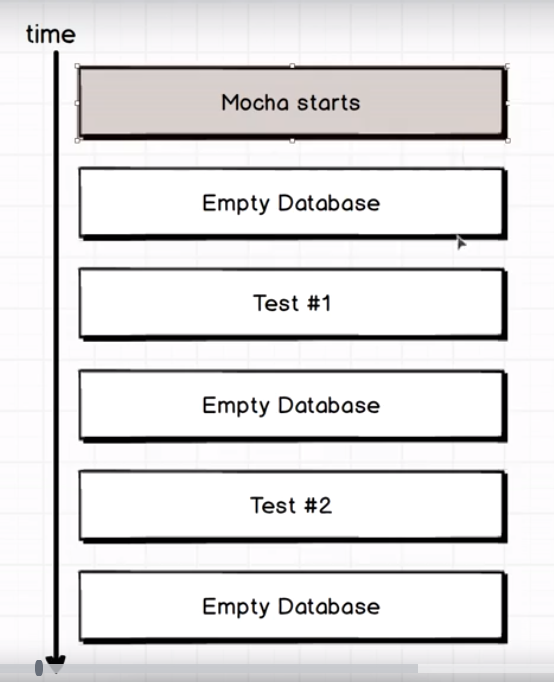
Output-

One thing to note is that if you do not have asserton in your it block. Mocha will consider that you have passed all tests.

In robomongo we can see that new db is created , then a new collection. In that collection our record is inserted. Otherthing is every time we run this script a new document will be inserted. So I want to ask you is do you thing in testing environment , this will a bad thing?

23)Dropping Collection

Yes, this will be problem. Here is why- lets assume that maybe we write a test where we insert a record into users collection and then maybe we do a search over that collection and say, hey does someone will name joe exists here? Well if you do not do cleanup to this collection of users that test will always succeed no matter what, even if our implementation is broken. So we need to make sure that we somehow do a little bit of cleanup every single time we run any test inside of our test suite. So here is the goal-



Everytime mocha strats up we empty our users collections. Then we run test, empty out collection of users and same again. This makes sure that no data from any other tests mess around with our test. Lets do it. Go to test\_helper.js. here we will add hook hook is a function that will be executed before any test gets executed inside of our test suit.

Code-

const mongoose = require('mongoose');

mongoose.connect(

"mongodb://localhost:27017/users\_test", {useNewUrlParser: true }

);

mongoose.connection

.once('open', () => console.log('Good to Go'))

.on('error', (error) => {

console.warn('warning', error);

});

BeforeUnloadEvent(() => {

mongoose.connection.collections.users.drop();

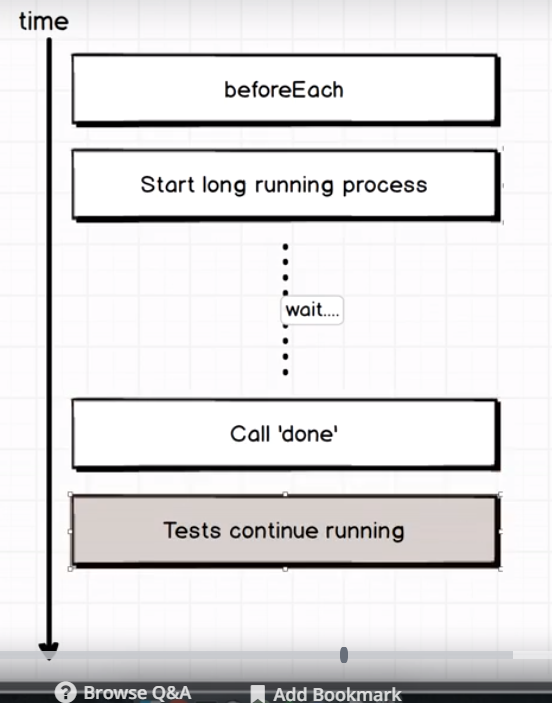
});

Before each is a hook. So in diagram, we will clear db before test runs. So this hook will be executed before any test is performed. Lets ay we execute 3 tests, then this hook will be executed 3 times.

Now there is on ne last thing. When we make connection with db it takes some time. same idea applies to every single operation that we take on collection as well. Its an operation that takes some amount of time to execute. And so we need to make sure that since this takes some amount of time to run here, we pause entire testing environment and say wait moka we are doing this long running thing, right here. Don’t actually run any more tests until this operation is complete. So lets figure out how we are going to tell to take little bit of a pause while are attemting to drop all of our users inside users collection.

24)Mocha’s done callback

We will take help of done callback in moka to solve problem that we discussed in last lecture. Done is a callback that is provided to us automatically by moka. After we call done callback, our test will contine to run.



So drop accepts a callback which will be executed once it has done dropping the collection.

Every single function that we write inside of moka that we hand off to beforeEach or it ot describe , gets called with done callback. So we pass done to function that we pass to beforeEach. Code-

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

mongoose.connect(

"mongodb://localhost:27017/users\_test", {useNewUrlParser: true }

);

mongoose.connection

.once('open', () => console.log('Good to Go'))

.on('error', (error) => {

console.warn('warning', error);

});

beforeEach((done) => {

mongoose.connection.collections.users.drop(() => {

done();

});

});

Now we are not making assertions in our test cases. We will fix it in coming lectures.

25)Mongoose’s isNew property

Here we will make assertions to ensure that joe is saved successfully.

Now we want to make assertion when record is saved. It is going to take some time a sthis is db related opreration. here we will use property provided by mongoose on model instance. When ever we have created a model and it is sitting inside nodejs workspace and has’nt been saved in db, mongoose will automatically place a flag on the model instance called isNew. If the record has not been saved to our database isNew will be equal to true.

When we saved model instance to db, isNew flag will be false. So we us ethis flag in create\_helper.js-

const assert = require("assert");

const User = require('../src/user');

describe("Creating records", () => {

it("saves a user", (done) => {

const joe = new User({

name: "joe"

});

joe.save()

.then(() => {

// has joe been saved successfully

assert(!joe.isNew);

done();

});

});

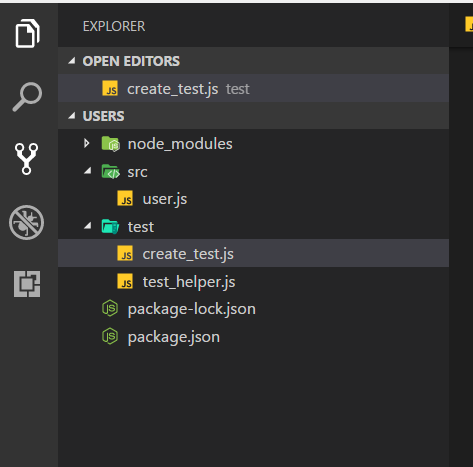
});

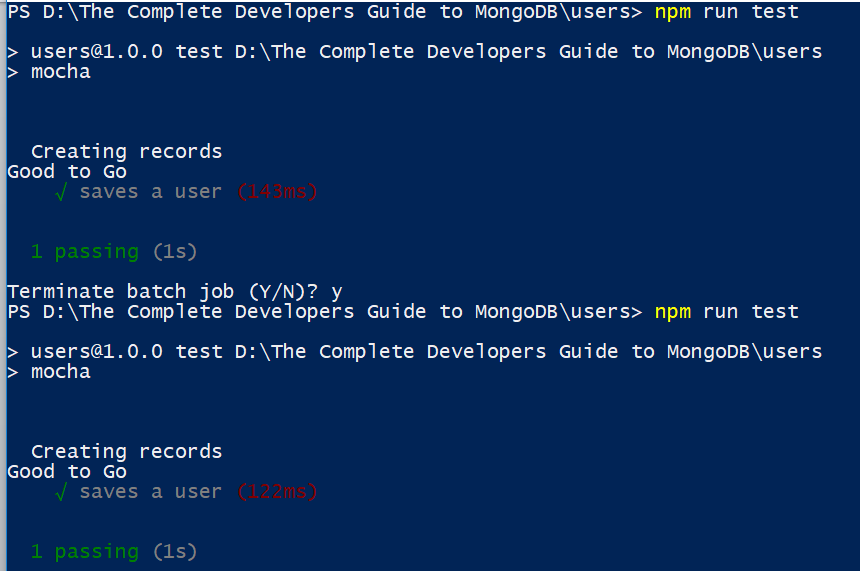
Now last thing remains we need to tell mocha to pause execution till assertion is made in this test. So we will use done callback. We used done callback for same purpose in test\_helper file, i.e to pause execution (to move to next test case) while database is cleared.

Done is avalaible in every single it block and beforeEach statement we use in mocha. If we are not using it, then we do not need to pass it as a argument to it block and beforeEach statement. But we you pass it make sure you are using it.now go to to user folder in cmd and run this command-

**npm run test**

folder structure is –

  
we will see that our test will pass. Output –



Now lets fail our test case. Use this code-

.then(() => {

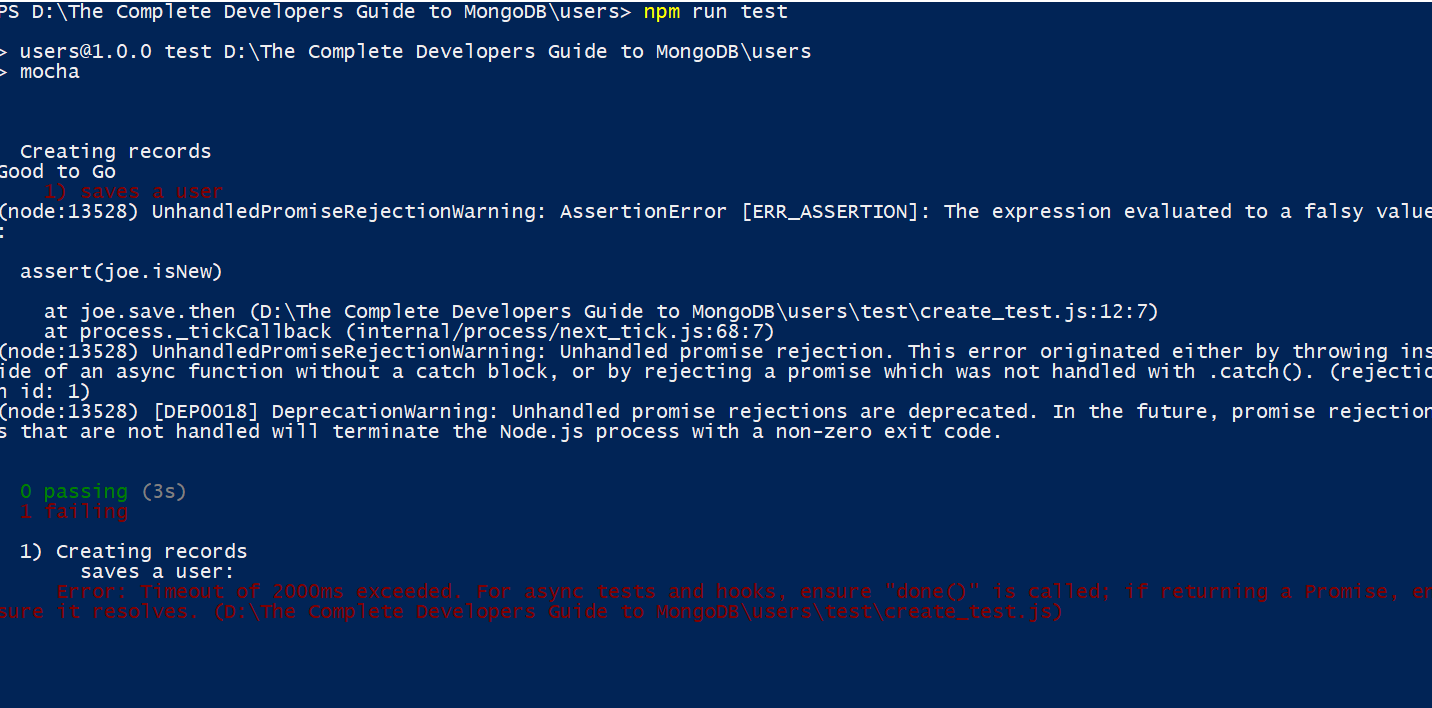
// has joe been saved successfully

assert(joe.isNew);

done();

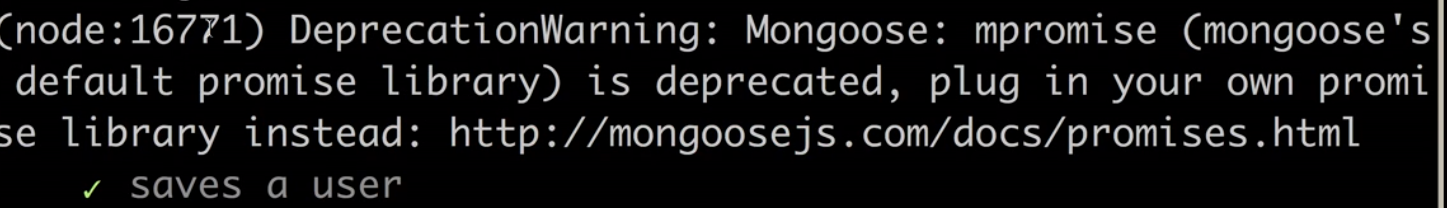
});

Output-



It has time span of 2000 ms exceeded, it means hey we got to our assert statement and it failed and we never reached the done statement. So our entire spec fails.

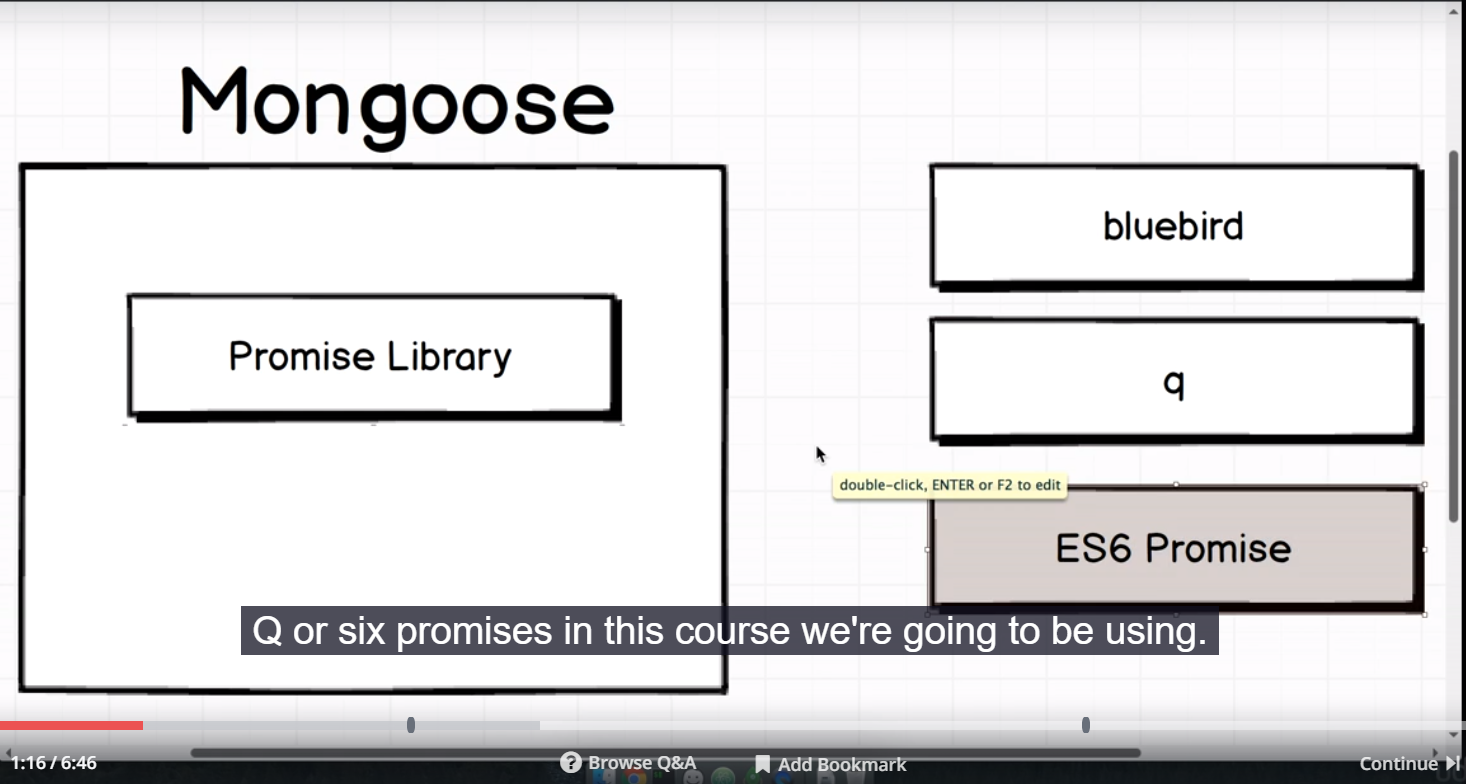
In success case, in video we saw this warning, but I have not seen it on my system-



In next lecture we will look into it.

26)Default promise library

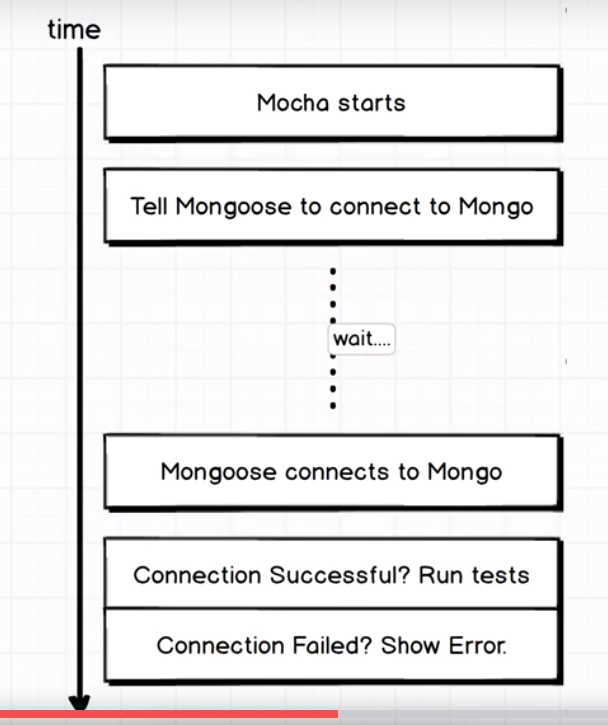
By default whenw e use mongoose, it has a default promise library plugged into it. We saw that save function(or any db operation function) retuned a promsie. This built in promise implementation this, mpromise(see error) thing which is plugged into mongoose. So this message is saying , we have default implementation of a promise library but for whatever reasons we do not want you to use it. So instead w ehave option to plug in our own promise library.

right hand shows popular promise libraries. Herew e will be using ES 6 promise. Go to test\_helper.js file. add this line-

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

After you import mongoose, add second line. global.Promise is refrence to Es6 implementation of promises inside nodejs environment. Now run your tests that warning will be gone. Now to test\_helper.js file. there is one issue around open that we kind of started to address but never really wrapped up.



One thing we really never did was we never tell moka to run our test. All our tests are working right now, but this is more about handling the case in which may be for whatever reason it took longer that we expected to connect to mongoose and maybe test suite moka started to run our test too soon(before connection is made). Code-

Test\_helper.js-

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

mongoose.connect(

"mongodb://localhost:27017/users\_test", {useNewUrlParser: true }

);

mongoose.connection

.once('open', () => console.log('Good to Go'))

.on('error', (error) => {

console.warn('warning', error);

});

beforeEach((done) => {

mongoose.connection.collections.users.drop(() => {

done();

});

});

So to accommodate for tat case to make sure that we were always going to be in clear 100 percent of time we are not just relying on mongoose connecting to mongo really quickly. We are going to wrap our connection statement with a before call. We have used beforeEach . this time we will use slightly different hook, before. This also will be called with a function that we write, the difference between before and beforeEach is that before is executed only one time for all tests ,your entire test suite. So we move our conncetion statements inside before. Now this connection is going to take some time, so again we want to say to moka, hey just hold up for a second, pause execution and wait until we have successfully connected to mongo. So again we will refrence done callback. So call done in case connection is successful. Code-

const mongoose = require('mongoose');

mongoose.Promise = global.Promise;

before((done) => {

mongoose.connect(

"mongodb://localhost:27017/users\_test", { useNewUrlParser: true }

);

mongoose.connection

.once('open', () => done())

.on('error', (error) => {

console.warn('warning', error);

});

});

beforeEach((done) => {

mongoose.connection.collections.users.drop(() => {

done();

});

});

27)Test setup for finding Users

In test folder create a new file, reading\_test.js we will come back and will do couple of more tests with create but for now lets move to tests for finding users.

Here before we find user w ehave to insert user first because we clear database before running each test case. For this we will use beforeEach. Then we use done callback to continue execution. Code-

reading\_test.js –

const assert = require('assert');

const User = require('../src/user');

describe('REading users out of database' ,() => {

let joe;

beforeEach((done) => {

joe = new User ({name: 'Joe'});

joe.save()

.then(() => {

done();

});

});

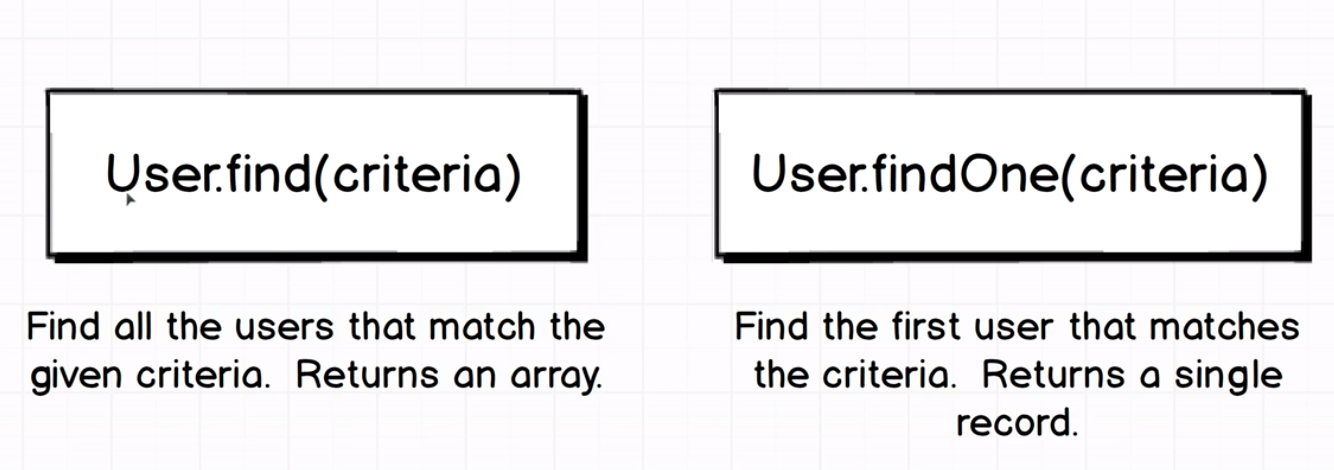
it('finds all users with aname of joe',() => {

});

});

28)Making Mongo Queries

To make query we are going to make use of class function or class method. We will make use of these 2 methods-



One thing to note here is these methods are tied to our class of user. Because w ecreated user model, we have access to these methods. The difference between these 2 is findOne returns a single match. While find returns an array of matches.

Note that we call these query methods on User class, not on instances of this class.

Then we use find method, now find methods like all our db operations is going to take time, so we use done callback. We call done callback to tell moka that tis test is going to take some time, once we call done, you can go onto next test. Code-

Reading\_test.js-

const assert = require('assert');

const User = require('../src/user');

describe('Reading users out of database' ,() => {

let joe;

beforeEach((done) => {

console.log('nested before each');

joe = new User ({name: 'Joe'});

joe.save()

.then(() => {

done();

});

});

it('finds all users with name of joe',(done) => {

User.find({ name: 'Joe'})

.then(users => {

console.log(users);

done();

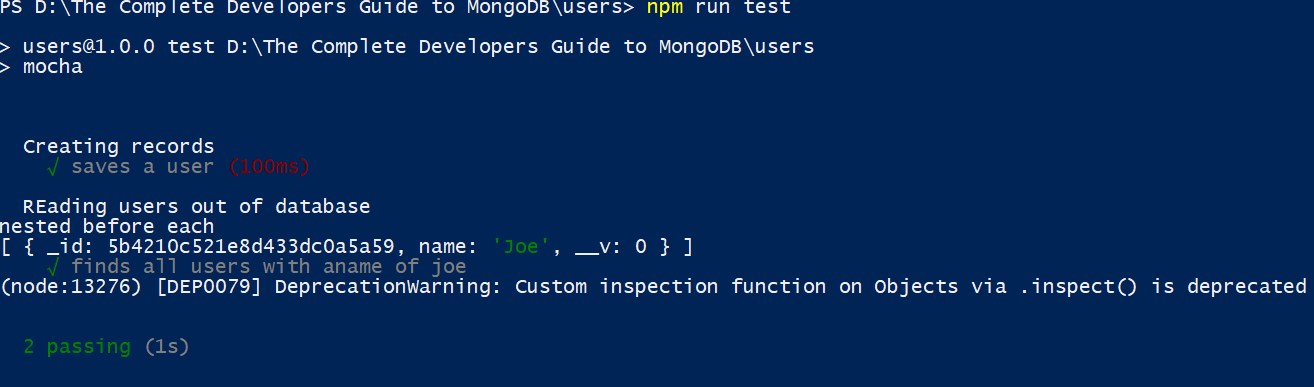
});

});

});

Go to users folder in cmd. And run- **npm run test**

Output-



Here we can see that before each callback for reading records is executed only for tat describe.

29)The Id proeprty- a Big gotcha

I outpt of first lecture we can see that we got our result back. Now how are we going to make assertion that result that we got back is same joe that have created in beforeEach statement, you can say that both have name of joe. You can say that it will have name of joe, but we want to find exeact user that we hae craeted. This would be great use case of using the Id property that very mongo records gets assigned. Nte, on ething when we created the instnace of user object and we have not created it, mongoose has assigned it a \_id property. When this object is saved , it is saved with this \_id. We will use this id to write assertion. But if we write our assertion like this-

assert(users[0].\_id === joe.\_id);

it will fail, because \_id proeprty of document is not raw string. It is object of type ObjectId.

Here we are comparing 2 objects and we know how that works. To solve this problem we need to get strings from \_id proeprty. We do this by toString method. Code-

const assert = require('assert');

const User = require('../src/user');

describe('REading users out of database' ,() => {

let joe;

beforeEach((done) => {

console.log('nested before each');

joe = new User ({name: 'Joe'});

joe.save()

.then(() => {

done();

});

});

it('finds all users with aname of joe',(done) => {

User.find({ name: 'Joe'})

.then(users => {

assert(users[0].\_id.toString() === joe.\_id.toString());

done();

});

});

});

Now we can see that our test cases are passed.

30)Automating tests with Nodemon

Everytime we have to run our test, we need to run command

**npm run test**

this is tiresome. Go to package,josn file, here we have script section-

"scripts": {

"test": "mocha"

}

It emans whenever we run **npm run test** we are running **mocha**  command. Now what we want to do is we want our test to run automatically for us any time that we change any file in our project. We are going to make use of tool called nodemon. Nodemon will watch our project directory for us. Anytme any file chnages, it will automatically rerun our tests.

If you have background with moch you can say that same thing can be achieved if we us this

"scripts": {

"test": "mocha --watch"

}

Reason we are not going to use that is that –watch sometimes has a litle bit of iffy functionalty or compatibility with mongoose. Inetead we will make use of nodemon which completely resarts mocha each time , a file is changed. We have already install moch as our dependencies. So we will replace our moch script. Code-

"scripts": {

"test": "nodemon --exec 'mocha -R min'"

}

When ever nodemon sees any change in our file, I want to run my moka command. So we use **–exec** , followed by command we want to execute. We have’t seen **–R** and **min.** these are for formatting out test output. Both of these are going to condense down our test output. These are also going to clear any output from the previous time the tests ran.

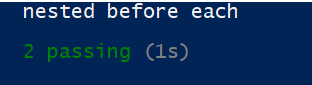
Ave file and run – **npm run test.** Now we will just see how many test haves passd, we wnt see descriptio of all tests. For some reason, I was getting error so I tried this-

"scripts": {

"test": "nodemon --exec \"mocha -R min\""

},

Output-



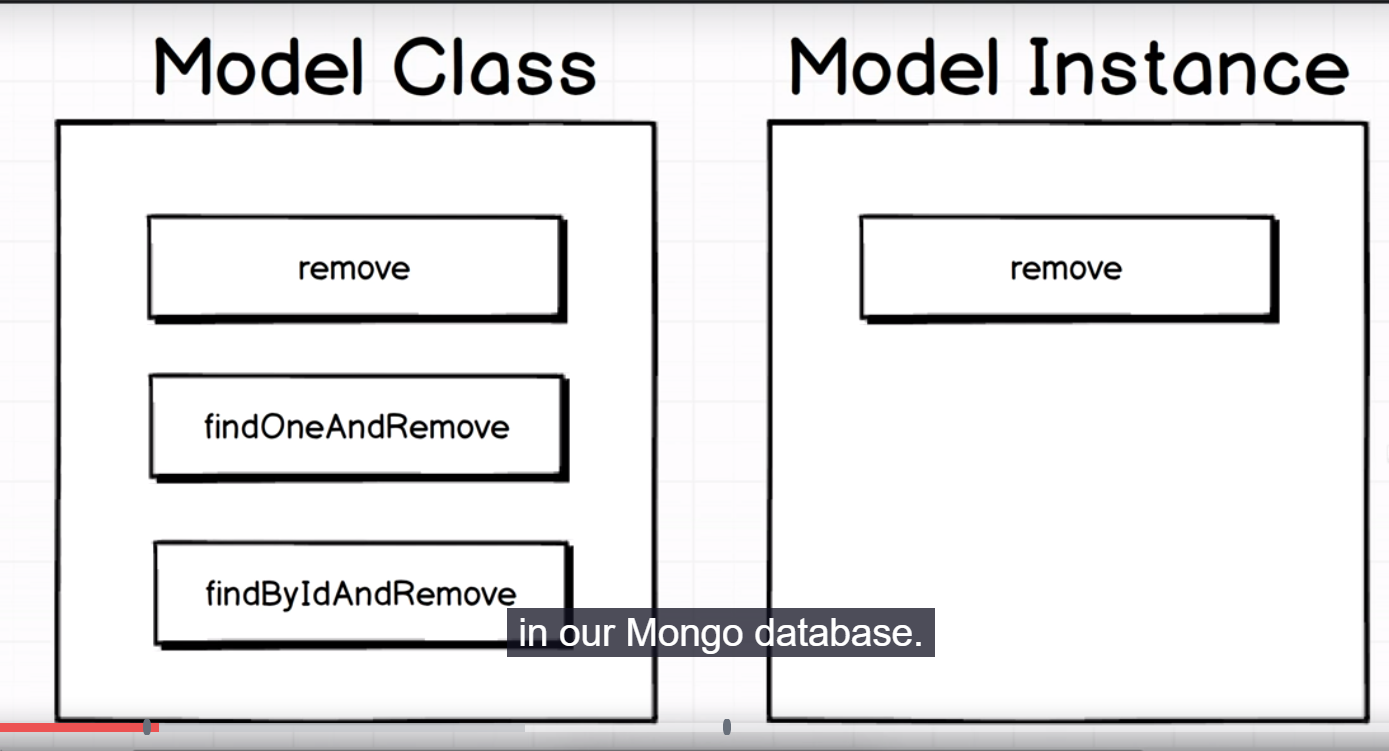
31)Finding Particular Records

Here we will use findOne method in our test case. We will find by \_id. Here we dnt have to use ObjectId because mongoose knows exactly how to work with that objectId thing. So joe.\_id is an Objectid object but mogo’s knows how to deal with that. So we can just say there is entire objet id, go find a record with an idea that looks like this.

32)The many ways to remove records

We will skip updating users and we will go to deleting users first. Create new file delete\_test.js. here we will create users and then figure out difrent ways to delete them.

There are many ways to achieve the exact same effect in mono and mongoose. Deleting a user is going to be fntastic example of that, that I reason we skipped update and showing delete here. So these are different ways-



Note, monggose does not use terminology delete, it uses remove. On let hand side all methods take argument to find which document to delete, but on right hand side we call this method directly on instance. So there is no need to find document, there fore there is no need to pass arguments to remve. Remove(right hand side) is useful when we have direct refrence to particuar record inside of our program. So if we have like already fetched joe and we have like an instance of joe, great time to use remove.

Here we will write assertion for remove(right) method. First we create a document , then we remove it. Now how do we write assertion to check whether doc ws removed for not. We search document and if it is not found that means it was removed. Here we will use promise chaining. from first promise we return another promise, so we can chain then. second then will get value resolved by promise that we returned in first level.

Code-

delete\_test.js-

const assert = require("assert");

const User = require("../src/user");

describe('Deleting a User', () => {

joe = new User({name: 'Joe'});

beforeEach((done) => {

joe.save()

.then(() => done());

});

it('model instance remove', (done) => {

joe.remove()

.then(() => User.findOne({name: 'Joe'}))

.then(user => {

assert(user === null);

done();

})

});

it('class method remove', () => { });

it('class method findAndRemove', () => { });

it('class method findByIdAndRemove', () => { });

});

Run the test , they will all pass.

33)Class Based Removes

We used fancy setup for promise in last lecture we had 2 promise that need to be eexcuted sequencally.

Here we will use remove method on class. We use this when we want to remove bunch of dcuments, with a given criteria. Code-

const assert = require("assert");

const User = require("../src/user");

describe('Deleting a User', () => {

joe = new User({name: 'Joe'});

beforeEach((done) => {

joe.save()

.then(() => done());

});

it('model instance remove', (done) => {

joe.remove()

.then(() => User.findOne({name: 'Joe'}))

.then(user => {

assert(user === null);

done();

})

});

it('class method remove', (done) => {

User.remove({name: 'Joe'})

.then(() => User.findOne({name: 'Joe'}))

.then(user => {

assert(user === null);

done();

});

});

it('class method findAndRemove', () => { });

it('class method findByIdAndRemove', () => { });

});

34)More Class Based Removals

Here we implemented the last 2 methids also. code-

const assert = require("assert");

const User = require("../src/user");

describe('Deleting a User', () => {

joe = new User({name: 'Joe'});

beforeEach((done) => {

joe.save()

.then(() => done());

});

it('model instance remove', (done) => {

joe.remove()

.then(() => User.findOne({name: 'Joe'}))

.then(user => {

assert(user === null);

done();

})

});

it('class method remove', (done) => {

User.remove({name: 'Joe'})

.then(() => User.findOne({name: 'Joe'}))

.then(user => {

assert(user === null);

done();

});

});

it('class method findAndRemove', (done) => {

User.findOneAndRemove({name: 'Joe'})

.then(() => User.findOne({ name: 'Joe' }))

.then(user => {

assert(user === null);

done();

});

});

it('class method findByIdAndRemove', (done) => {

User.findByIdAndRemove(joe.\_id)

.then(() => User.findOne({ name: 'Joe' }))

.then(user => {

assert(user === null);

done();

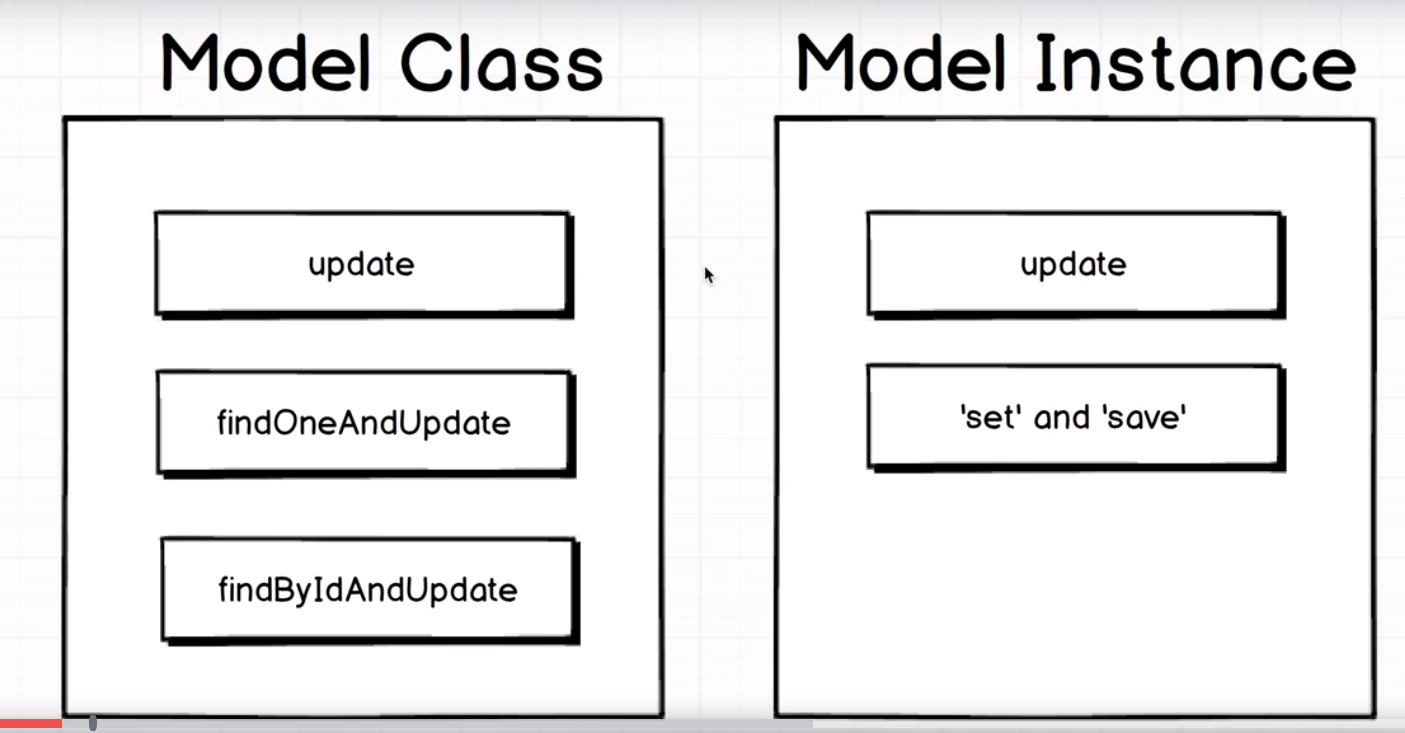
});

});

});

35)The Many Ways to Update Records

Updating is where a lot of complexities come into play. Different ways to update-



Model instance methods are simple but with class methods , things start to get intersting. So create a new file , update\_test.js.

First we will see set and save , that is called on model instance. It is used when we want to update a single property , like here we are updating name. we can print joe object on console before and after calling set to se change.

however calling set does not update dataabse, this update is solely in nodejs code. To perisist these chaages to db , we have to call save method. Save can be used when we have used set on some proeprty. We can use it like that, we can update proeprty many times and then at last when we are done with updating, we can use save method.

While writing test for update, we need to use different approach then we used for delete and save. We should not only be testing there is no record with name joe, but we should also be testing that there is record with name of alex.

So we pull up all users to do that we pass mepty object to find method.code-

Update\_test.js-

const assert = require("assert");

const User = require("../src/user");

describe('Updating recods', () => {

let joe;

beforeEach((done) => {

joe = new User({name: 'Joe'});

joe.save().then(() => done());

});

it('instance type using set n save', (done) => {

console.log(joe);

joe.set('name', 'Alex');

joe.save()

.then(() => User.find({}))

.then(users => {

assert(users.length === 1);

assert(users[0].name === 'Alex');

done();

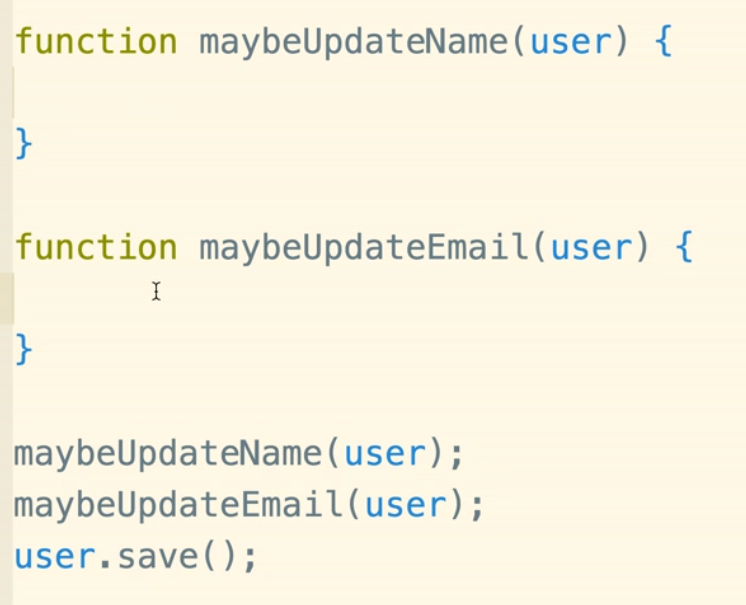
});

console.log(joe);

});

});

So the whole idea behind set and save is that w emight want to update several different proeprties in different steps and then only after making all those updates, we will make final save. This methodology for updating a record in this kind of piecemeal fashion is best when you want to update a couple of different properties but you might want to do so in different steps. Ex-



Here we will not call save in those 2 functions, we have these functions which have all the logic to decide whether we should update this singukar proeprty or not. When everything is done , then we call save. This approach is much better than updating database in both functons.

37)Model Instance Updates

Here we update the name of single user. We will call update on model instance. In delete file we hve to copy paste 2 then statements for reach tests. Instead of doing same copy paste here, we will make helper fuction that will make this assertion for us.all that we have to do is to call this assertion function.

We define a function assertName. We assume that it is going to be called with result of save or updte or somehing operation. So basically it will be called with an operation that results in the records being updated inside of our users collection. We call this thing operation., it is like we will be calling assertNamefunction and passing joe.save() as argument. The operation will be promie so we can chain .then calls to it. Then we cut all lines from joe.save-

joe.save()

.then(() => User.find({}))

.then(users => {

assert(users.length === 1);

assert(users[0].name === 'Alex');

done();

});

And add them to this operation argument inside aseertName function. like this-

function assertName(operation) {

operation.then(() => User.find({})).then(users => {

assert(users.length === 1);

assert(users[0].name === "Alex");

done();

});

}

In it block we call joe.save apassit as argument to assertblock. We laso pass done as second callback. This ake sure when we call done in assertName, we are talking about same scope.

Then we make assetion for update on model instance.code-

Test\_update.js-

const assert = require("assert");

const User = require("../src/user");

describe("Updating recods", (done) => {

let joe;

function assertName(operation, done) {

operation.then(() => User.find({})).then(users => {

assert(users.length === 1);

assert(users[0].name === "Alex");

done();

});

}

beforeEach(done => {

joe = new User({ name: "Joe" });

joe.save().then(() => done());

});

it("instance type using set n save", done => {

console.log(joe);

joe.set("name", "Alex");

assertName(joe.save(),done);

});

it("A model instance can update", (done) => {

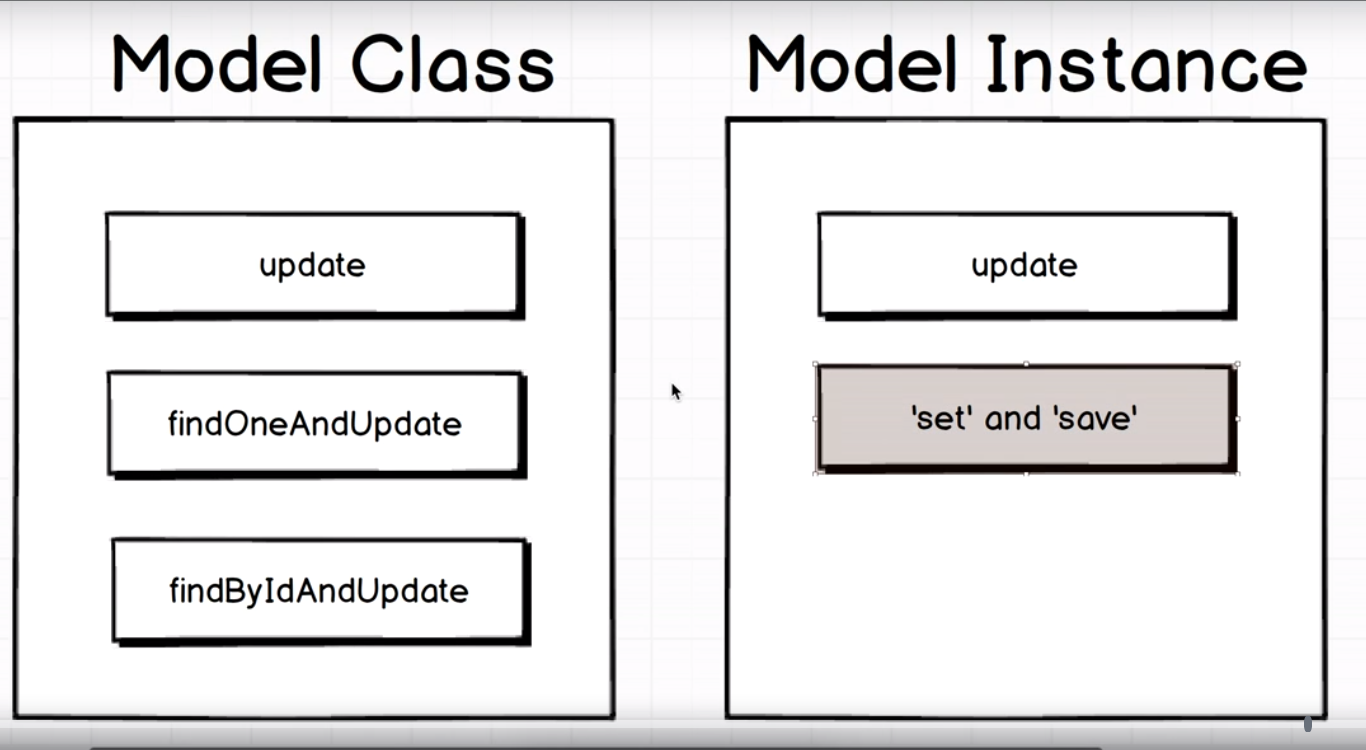
assertName(joe.update({name: 'Alex'}),done)

});

});

Here update functions updates the object and saves it into database in one go. These instances methods are useful if we have one particular record, laready loaded up. i.e we have already reached into our database, we have pulled out joe and now we want to make update and save it.

38)Class based update



We have covered right hand side. Now we will see class based methods for update. In practice all three of these updates are ging to work very similar to how remove helpers work as well. So update will update all records like remove will remove all records that mach criteria. Sam ething with rest 2. We said tht things will be intersting when we move to calss based updates. So first we will use these functions in easy way and then we will talk about how they can be kind of enhanced and used in a much more advanced fashion as well.

Code-

Test\_update.js-

const assert = require("assert");

const User = require("../src/user");

describe("Updating recods", (done) => {

let joe;

function assertName(operation, done) {

operation.then(() => User.find({})).then(users => {

assert(users.length === 1);

assert(users[0].name === "Alex");

done();

});

}

beforeEach(done => {

joe = new User({ name: "Joe" });

joe.save().then(() => done());

});

it("instance type using set n save", done => {

console.log(joe);

joe.set("name", "Alex");

assertName(joe.save(),done);

});

it("A model instance can update", (done) => {

assertName(joe.update({name: 'Alex'}),done)

});

it("A model class can update", (done) => {

assertName(User.update({ name: 'Joe' }, { name: 'Alex' }),done);

});

it("Model class can update one record", (done) => {

assertName(User.findOneAndUpdate({ name: 'Joe' },{name: 'Alex'}), done)

});

it("A model can find a record with an Id and update", (done) => {

assertName(User.findByIdAndUpdate(joe.\_id,{ name: 'Alex' }), done)

});

});

First let see update on class. First we select users based on criteria. As a second argumnet we pass the updated proeprties.

findOneAndUpdate, we are finding a single record then we are updating it. a\thne we have last method.

So these were easy ways, in next section we are going to figure out more complex methods of implementing updates on collection of records.