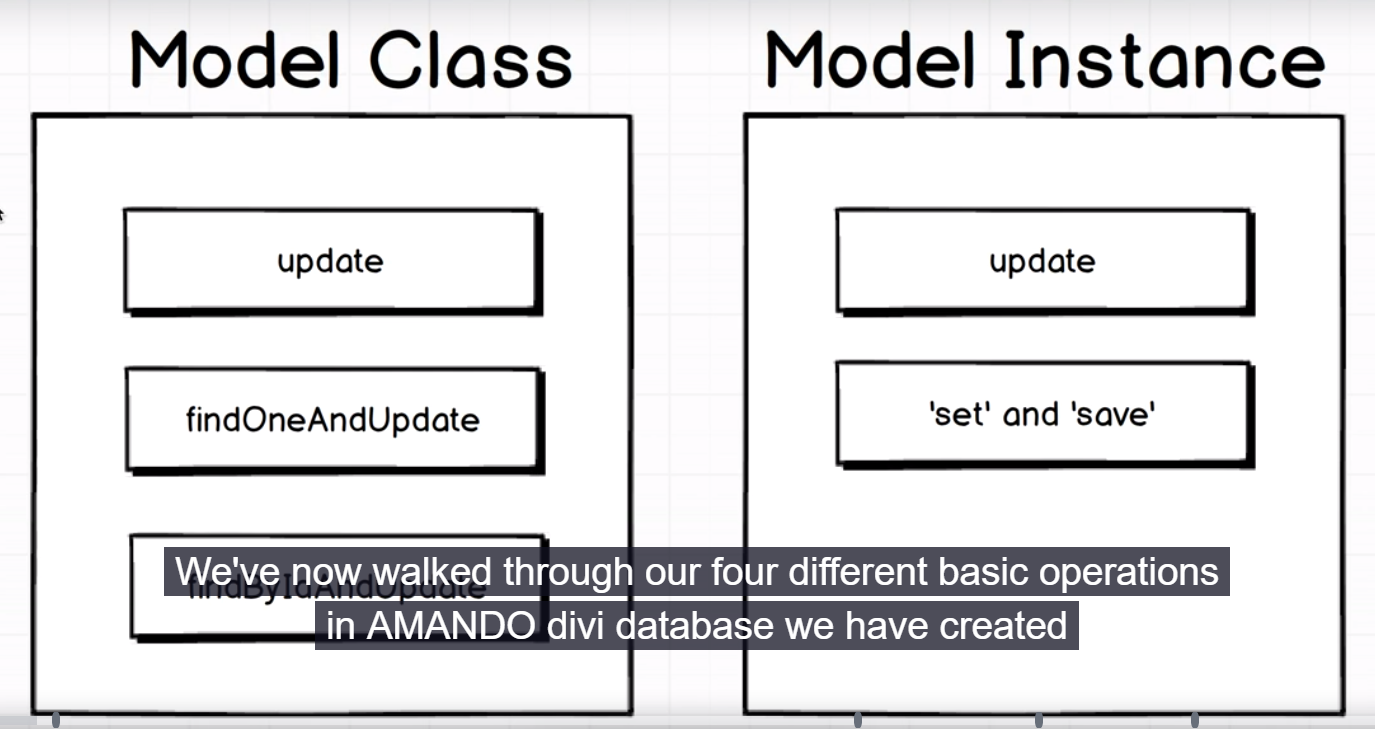
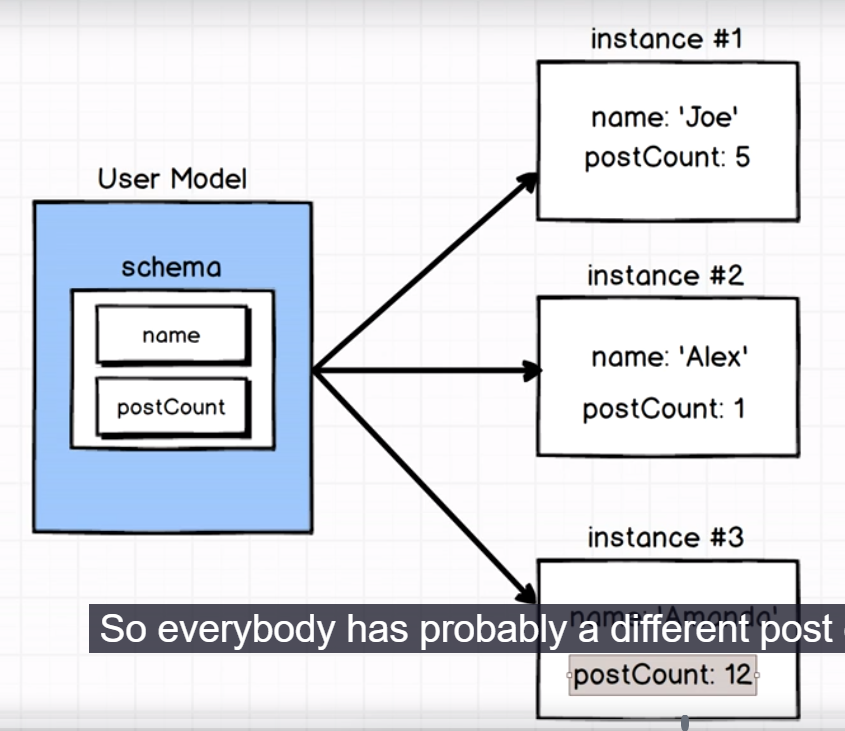
39)Update operators

Just covered simple use cases of update functions of model class. 

Now we will see their advnaced useage. So this is where we are going to strat learn a lot more about performance inside of mongo DB. lets give you a idea, where we are going to. I want to add in anew proeprty to our user model. I want to add new Proeprty call postCount. This gives us the count of number of blog posts.



First we fo to user.js file and we added new property to our schema-

const UserSchema = new Schema({

name: String,

postCount: Number

});

Now our existing tests are still passing. I can have as many properties on my schema here as I would like, this tells mongoose that this property might exists and what type it should be. So it is ust helping us out, it is not enforcing any requirements for data that model actually has. Not until we start adding some validations to our model and start saying ok, I want to post count to be a number greater than 0 or something like that. Then I would see some test will start to fail.

Now I want to have a new update test, where I want to increment the user postCount. And here we want to use class based update methods. We want to update a group of users, so we cannot use instance based update methods. First we add postCount property to joe in udate\_test.js file. code-

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", postCount : 0 });

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)

});

it("A user can have their post count incremented by one", (done) => {

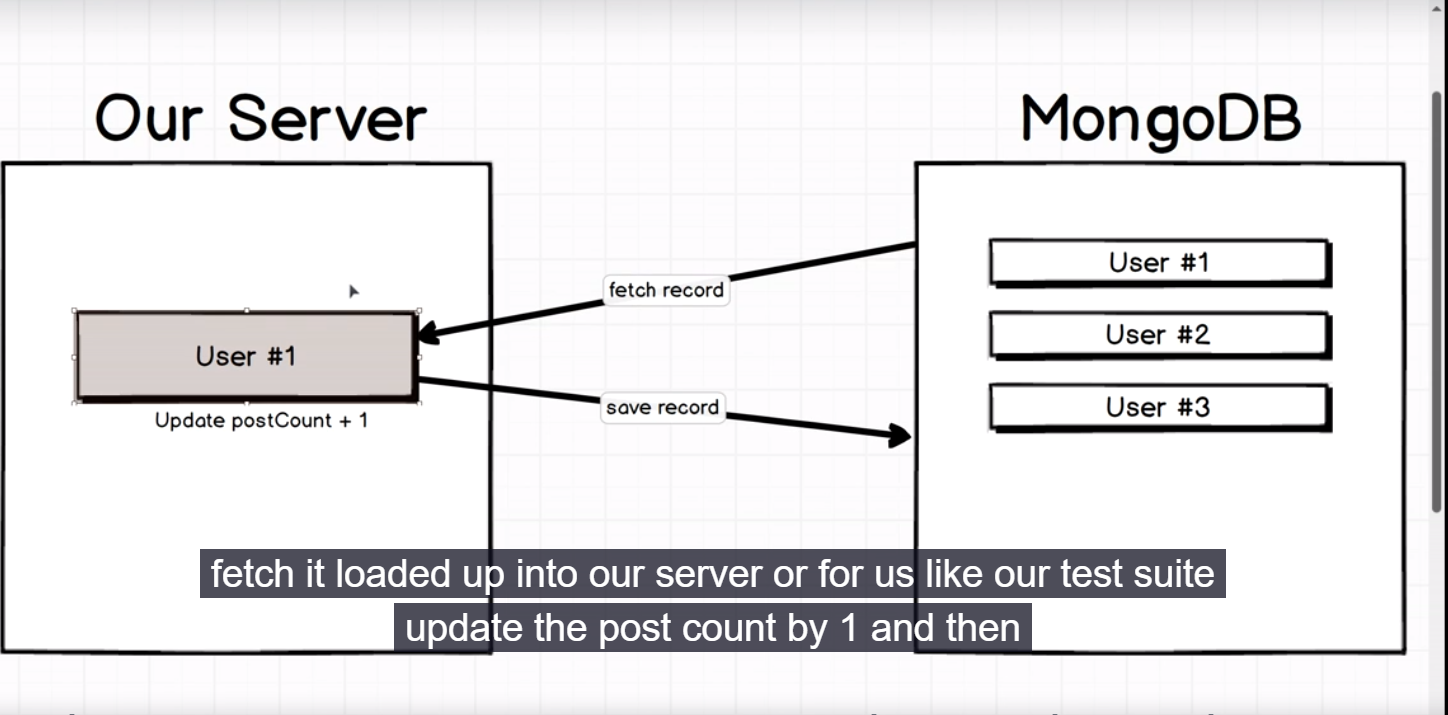
});

});

As initial count is 0, final count will be 1. So basically at some point of time, we want to write a test that says something like assert that count is 1. That’s where we are headed. So I want you to think , using all the tools that we have right now, that we have learned about making updates, how might we update every user or maybe every user with name of joe(which could me many), add one to each of their post count. Well you might say we can do it like that-

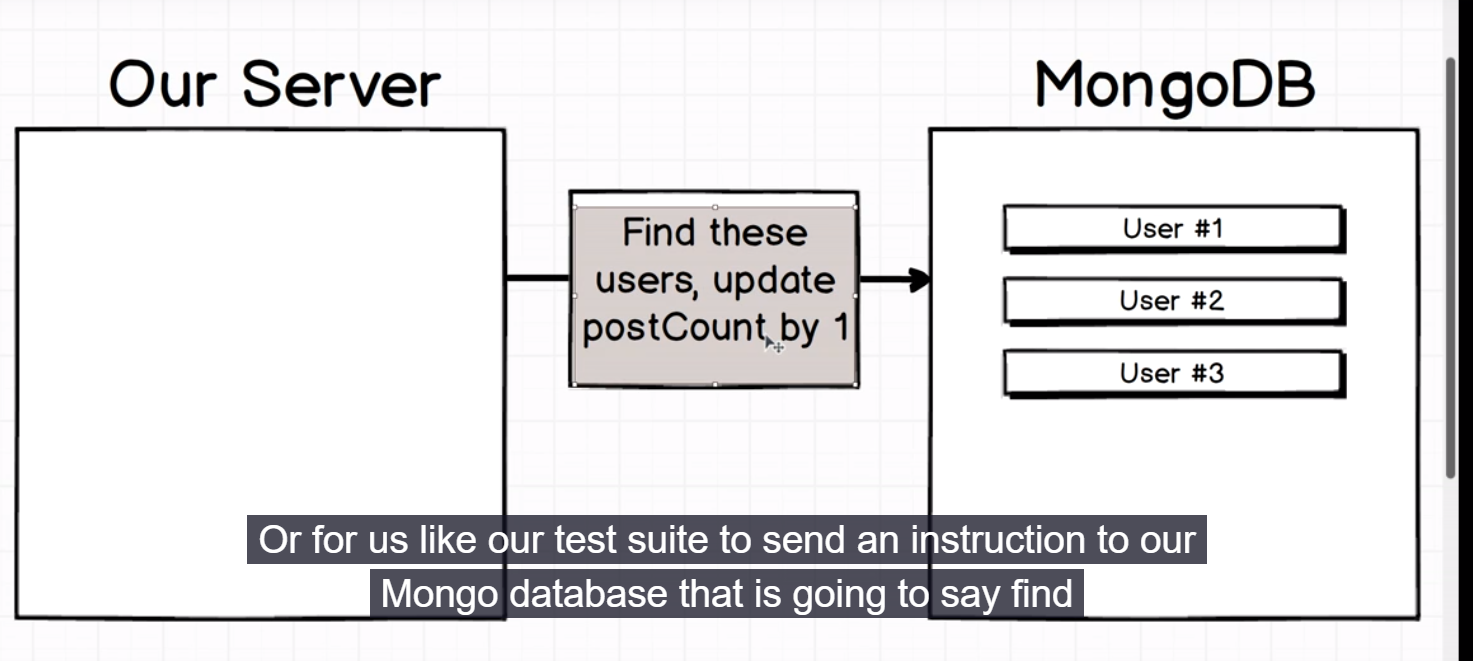
User.update({name: 'Joe' {postCount: 1});

But there is alittle bit of trap here. this will kind of lead you to wrong path. Herew e are incrementing the post count by 1, we are setting post count by 1. Then you might propose this solution-



Here we will do find for each user which meets a criteria like name , joe and then iterate through each user, look at their current post, increment it by 1 and then we will save it. This will work but it is not efficient.

Any time we can avoid loading data into our server from mongo is big win for us from a performance standpoint. So we are going to use a slightly better approach. We will use this approach-

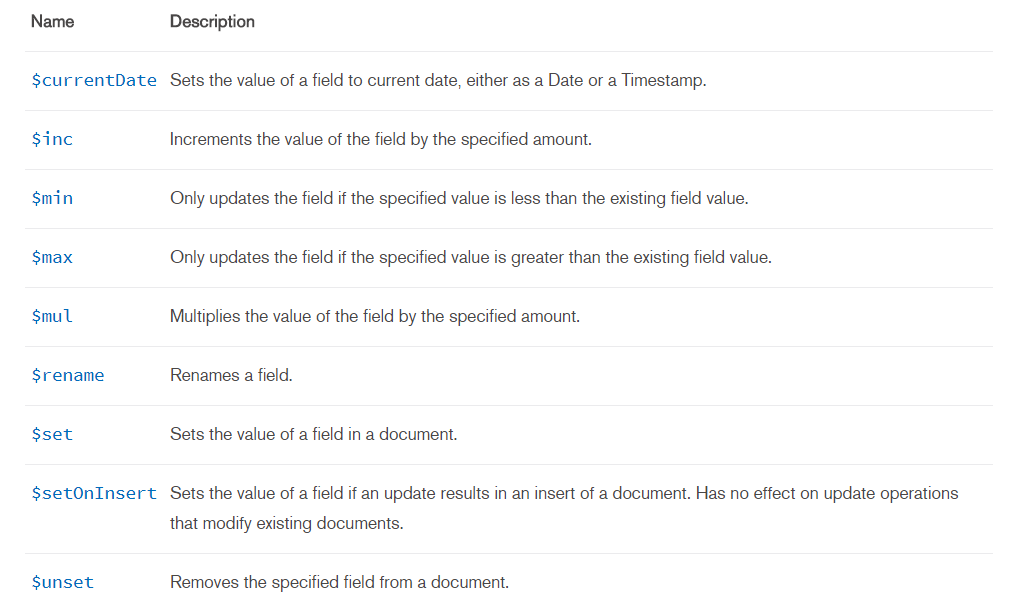


We are going to use our server(for us its out test suite) to send aninstruction to our mongo db , this isnctruction will say find these users and increment their postCount by 1. Here we are entirely relying on mongo to make these changes for us. This whole discussion is opening the way to what we call- **mongo update modifiers** and its defimatelt one of more challenging ideas or topics inside of mongos. Lets go to docs. Google this-

Mongo update operators

Open first link that is of official mongo db page.

This is collection of what we call as mongo update modifiers. We can use these modifiers or operators in conjuction with the update function inside all our different methods of updating a record. So it works for all update functions of model class. What these do is they allow us to send an instruction off to mongo and mongo will execute that instruction inside database. Reason that we use these operators is that they are fantastically efficient in performance compared to last approach, we saw in this lecture. Lets have look at some of them-



Set mean if we want to change some fields value to some other value. $min will allow us to make update only a value is less than some amount. $max also works like same.

In practice each of these operators are fantastically useful , whenever you want to change a lot of different records in one go. So if you are updating just a single record, you do not need to reach for these but if you want to update like 5,10, 15 records all at once, you should use these.

40)The Increment Update Operator

Code-

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", postCount : 0 });

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)

});

it("A user can have their post count incremented by one", (done) => {

User.update({name: 'Joe'}, {$inc: {postCount: 1}})

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

.then(user => {

assert(user.postCount === 1);

done();

});

});

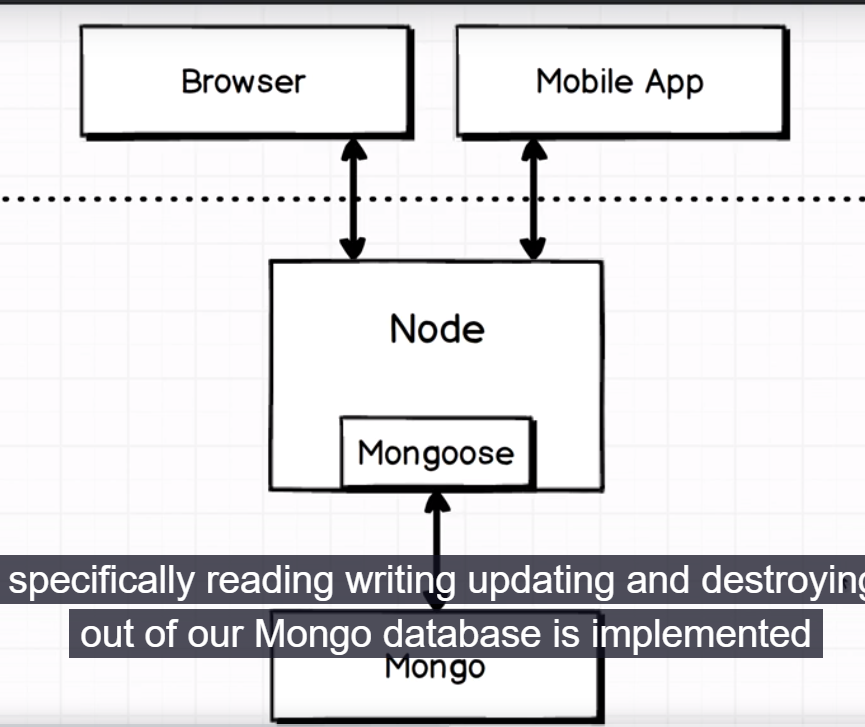
});

There is no decrement operator.so if you want to decrement, then pass negative value.

Set operator is also useful. It’s usually the case that we don’t get as much leverage out of the set operator even though I kind of mentioned it in particular because if you are just setting a value of a filed in document then we might as well use the update function as we saw earlier. So inc and mul are useful but there are some times when you want to check other ones as well.

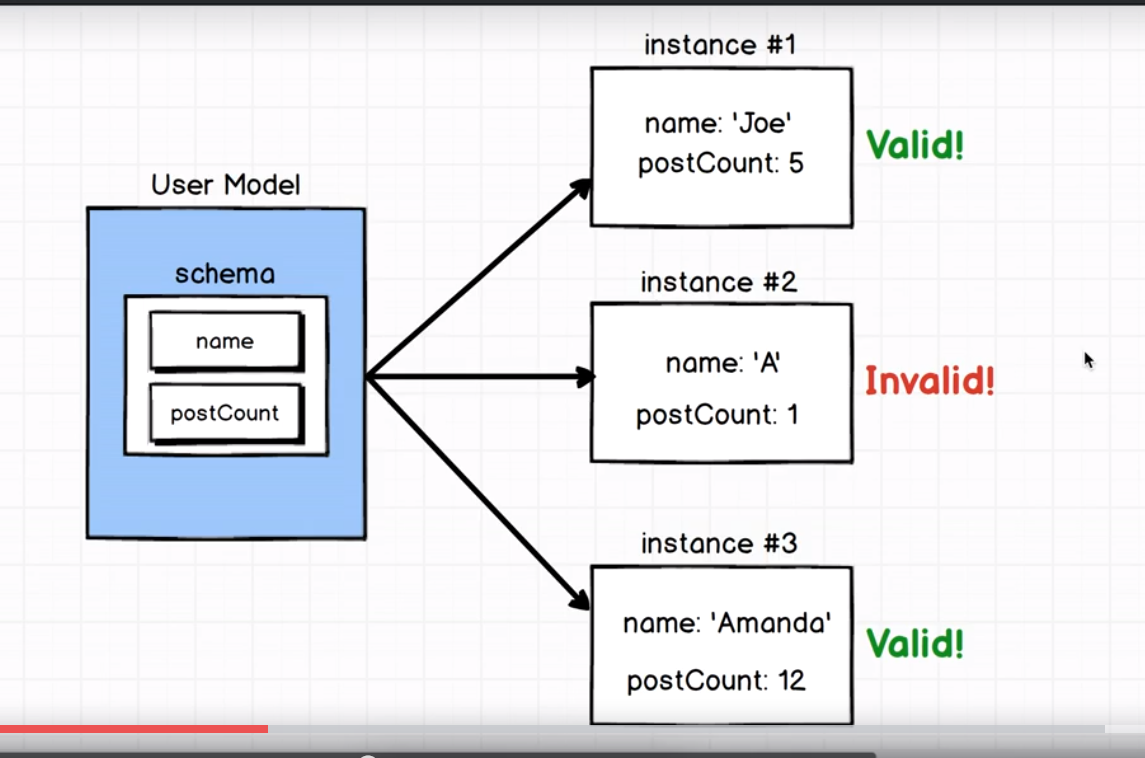
41)Validation of records

Now as we are done with basic crud operations, we can start to explore a lot more, what mongoose has to offer. And then after we are done with broad overview we can work on our next project. Lets reconsider what we are working with when we touched mongoose.



When we work with mongose we are working with the interface to mongo. Mongoose is all about giving us nicer API something that has advanced functionality to work with mongo. all of the operations that we have worked with so far (like, specifically CRUD operations), out of our mongo are implemented by mongoose but we could have just as easily reached directly to mongo and essentially had basically the same experience. What we are going to start talk about now is a lot more of advances features around mongoose that offer us great, awesome functionality around mongo, that is not implemented at all by default. First feature is validation.

Validation is process of evaluating whether or not properties of record are safe to save to our database.



We want to add validation to my model that user name needs to be longer than 2 characters. That explains valid and invalid in last pic. Once model instance is invalid, it won’t be allowed to be saved in database. The purpose of validation is to make sure that we do not insert junk data into database. This validation we are about to add in is solely implemented by mongoose, as we are using it atleast. As validation level we are using is 100% mongoose. Mongo does have support for validation but we are not directly working with it. Lets see validations. Crate new file called validation\_test.js there we will write a test to kind of verify that validation is working as it is expected to be. In file just write some boilerplate code.

42)Requiring Attributes on a model

Here we will write test case to see that model instances passes these 2 validations- it has username and username is greater than 2 characters. 2 separates rules , so we will write 2 separates tests. It means we will have 2 it blocks. First lets add validation to our schema. Right now , our schema in user.js is-

const UserSchema = new Schema({

name: String,

postCount: Number

});

So instead of telling that it should be string. We pass a full object and then inside this object we are going to add a bunch of different properties to describe not only the type of name because we still have to specify the type, but we are also going to specify some of the validation options that we want to apply to named property. Lets change it-

const UserSchema = new Schema({

name: {

type: String,

required: [true, 'Name is required']

},

postCount: Number

});

We pass a required flag. We assign an array to it, first value is true. It makes this field compulsory, then we have message to be displayed whenever a user name is not provided. We will revisit fie to provide length validation.

Now we go to validation\_test file. there we assign name undefined , so that any developer can understand that we name name to be undefined. Our test revolves around that. Then we do try to save user in db. I want to validate this model instance and see what is prodiced by this validation process. So we call **validateSynce** on user. We store this result in validationReult. This validation result object right here is going to have all the results of validating the user model. So if we have ton of proeprties wrong, there will be lot of properties on validation result that we can look at.

Name of function is validateSync, it is to differentiate it from a very similar but slightly different function called validate. Difference is that former is synchronous process. With validateSync ,We can store the result in variable like this-

const validationResult = user.validateSync();

validate works like this-

user.validate((validationResult) => {

});

So we pass a callback which will be called with validationResult and then inside this function we can do something with validation result. Reason we would have validate with callback function right here is to run any type of asynchronous validation we might want to have. Like we cn make request to api, check that last name does not exeist in db. etc. as we are not doing these things, we will stick with valudateSync. So lets print our validationResult. Code-

const assert = require("assert");

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

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

it('Requires a user name', () => {

const user = new User({name: undefined});

const validationResult = user.validateSync();

console.log(validationResult);

});

});

This gives us entire stack trace. When validation errors out it actually produces a error object. so this is error object, when js produces error object it is going to give us stack trace as well. Then we extract message property and place our assertion around it. Code-

Validation\_test.js-

const assert = require("assert");

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

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

it('Requires a user name', () => {

const user = new User({name: undefined});

const validationResult = user.validateSync();

// const message = validationResult.errors.name.message;

const { message} = validationResult.errors.name;

assert(message === "Name is required");

});

});

43)Validation with a validator Function

Now we will add advanced validation. We want name to be atleast 3 character.

So we add new property to schema, validate takes a object. so validate will be looked at to find some information about how this property should be validated whenever the validate function runs. We assign 2 properties to this object. one is validator which is a function, other is message, which is shown whenever validator function fails or whenever it returns specifically false. Validator function will be called by name itself. This function should return true or false. True means property is valid and false means it is not valid.

User.js-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const UserSchema = new Schema({

name: {

type: String,

validate: {

validator: (name) => name.length > 2,

message: 'Name ust be longer than 2 characters'

},

required: [true, 'Name is required']

},

postCount: Number

});

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

module.exports = User;

now lets move to our validation file. validation\_test.js-

it('Requires a user name longer than 2 characters', (done) => {

const user = new User({name: 'Al'});

const validationResult = user.validateSync();

const { message } = validationResult.errors.name;

assert(message === "Name ust be longer than 2 characters");

done();

});

44)Handling Failed Inserts

Here we will write test case to make sure that mongoose does not allow invalid records to be saved in database. validation\_test.js-

it ('Disallows invalid records from being saved',(done) => {

const user = new User({name: 'Al'});

user.save()

.catch(validationResult => {

const {message} = validationResult.errors.name;

assert(message === "Name ust be longer than 2 characters");

done();

});

});