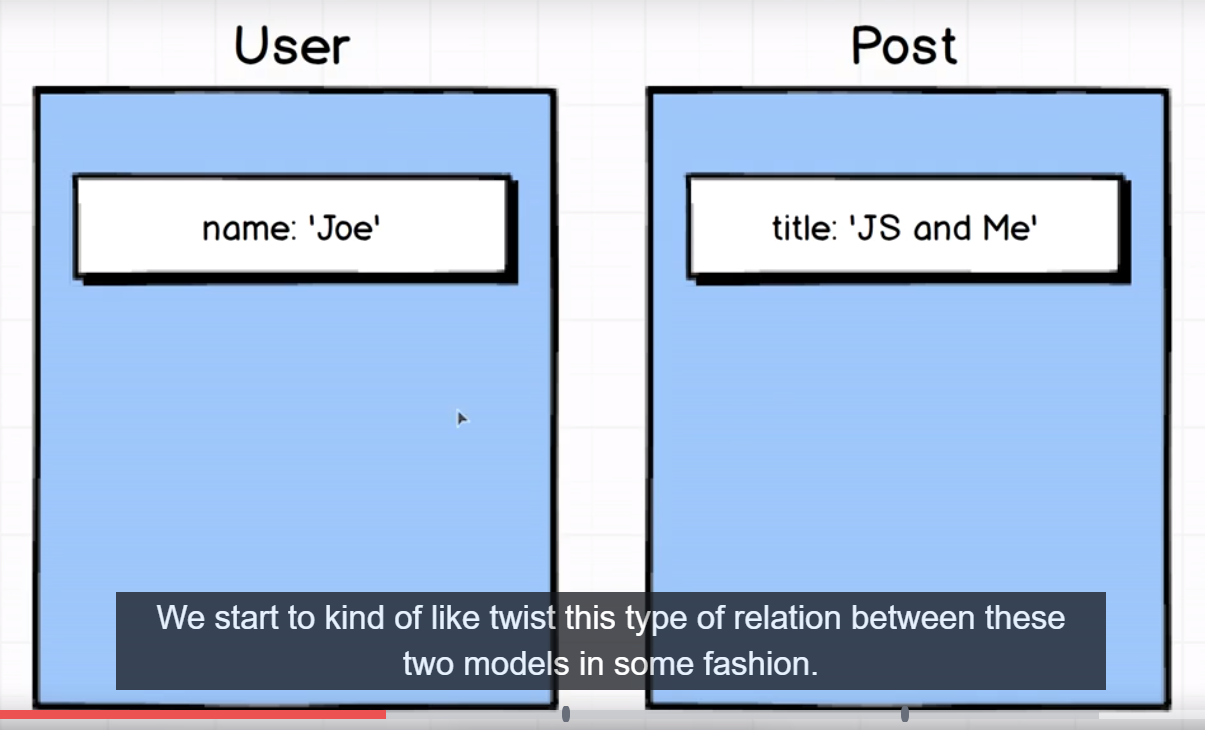
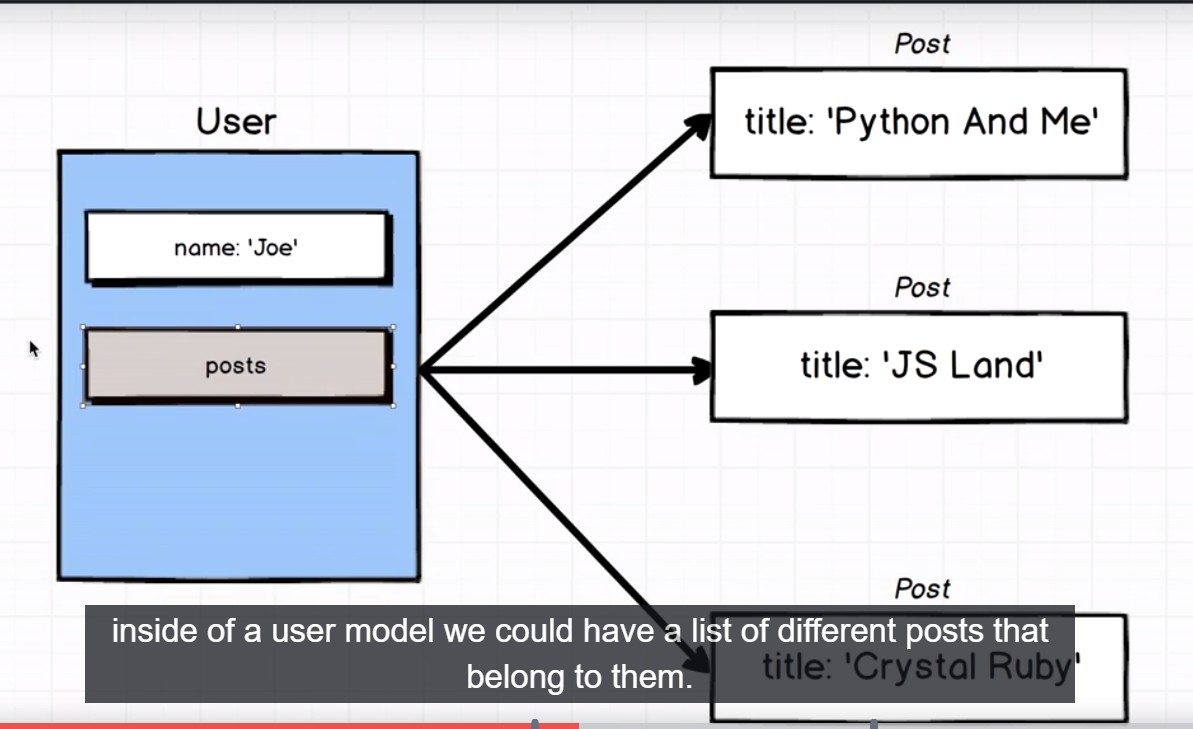
45)Embedding Resources in Models

Here we will add another type of model on top of user. We want to represent that user can have many posts. You can think that we will create another model. lets say we want to create model for posts. If we do this, then we will end up with something like this-

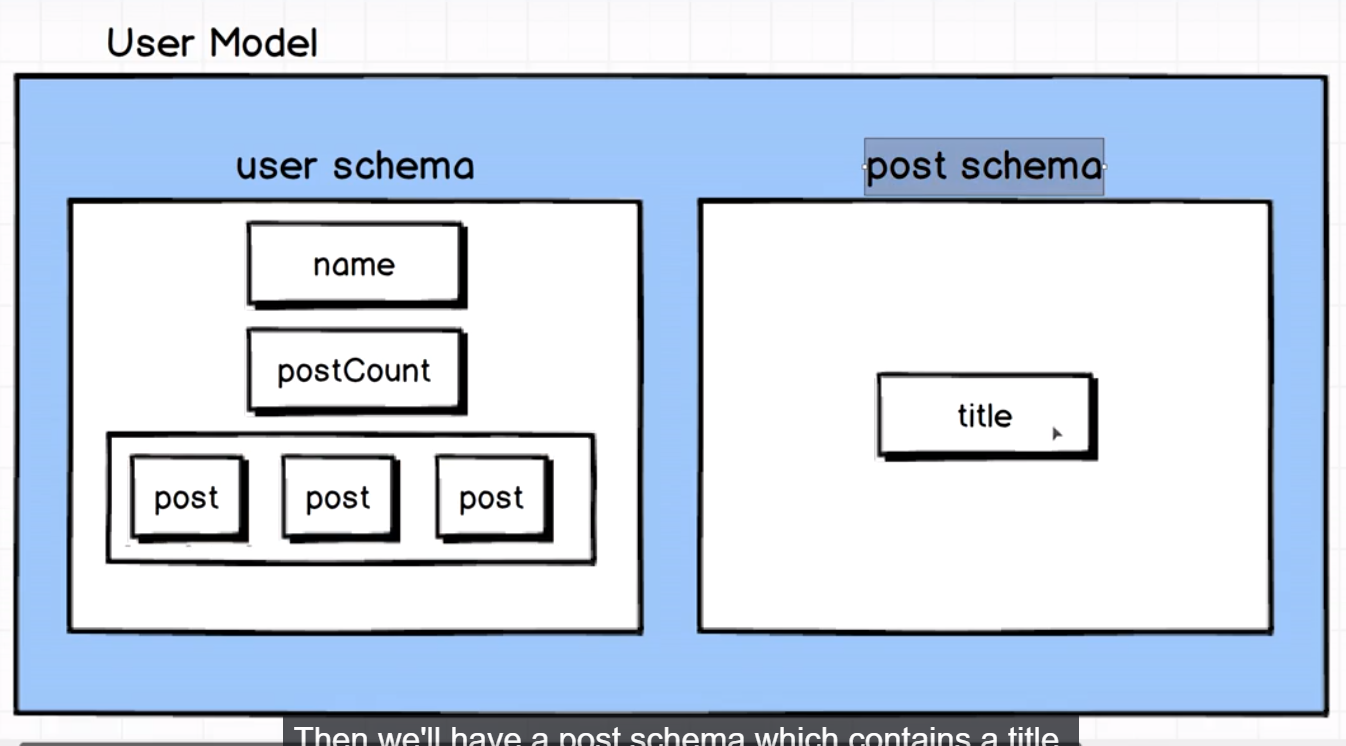


So these will be 2 completely different resources. This is the approach that we are looking at right here, would make a lot of sense inside of sql database. Inside of nosql db like mongo, we start to kind of like twist this type of relation between these models in some fashion. Important thing to mention here is that I said explicitly – a user has many posts. So a post is associated with exactly one user. With that in mind inside of our mongo database, a more reasonable type of data structure might be rather than to have a separate collection of users and separate collection of posts would be instead to still have a single collection of users and then inside of user model we could have a list of different posts that belong to them.



Here user joe has posts property which has 3 posts inside of it. Each of which have distinct title property. This concept of nesting records or associations rather than spinning off completely new collections is at the core of what makes mongo, mongo. We would talk more about possible design schema in future. For now lets walk down this path.

So thinking about this in a different fashion. This is our user model or diagram of user model-



Right now our user model has user schema. The suer schema has a name, postCount and I am suggesting that we add it to have ability to have an array of posts. Then we will have post schema which has title. I will repeat this several time, whenver we embed a document in this fashion, we will not create a separate model , I am specifically using the word model here to represent the nested resource. So we will not make post model. Instead we will make only a post schema. So we will write some amount of code to describe what apost inside of user should look like. But we will not make distinct mongoose collection or distinct mongoose model around a post. The reason for this is that mongoose models like our model are made to represents distinct collections. We are not going to have collection of posts. We are only going to have list of users or collections of users and inisde of that each user can possible have list of posts.

46)Nesting posts on users

What we discussed in last lecture is called idea of sub documents. User is document and post is sub document. Create file, post.js in src. Code-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = new Schema({

title: String

});

module.exports = PostSchema;

now we need to wire up this post schema with user model. User.js-

user.js-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = require('./post');

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,

posts: [PostSchema]

});

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

module.exports = User;

so here we say that user schema shoud have additional property posts and it should conatin array or lists of post schema. Mongoose is going to look at this property, it’s going to notice that its an array, so it knows that it should be list of records and then it will see that inside {} we have schema, so mongo is going to infer that ok. It should be nested subdocument or list of subdocuments of some kind.

47)Testing subdocuments

Create a file called subdocument\_test.js file in test folder. now, what we really want to test here. I think we should start off by saying you know, kind of figure out how do we just create subdocument and save it. This is our first test-

To add in ost we don’t really have to do something special. We just have to pass some amount of data that matches the structure that user model expects. So we create jow object like tis-

const joe = new User({

name: 'Joe',

posts: [{title: 'PostTitle'}]

});

Mongoose is going to see that we are providing the posts property here and it’s going to automatically attempt to apply post schema to this inner object right here.

{title: 'PostTitle'}

Note I said apply the post schema, what I really mean is that if we had written any validation into post schema or any kind of rules like that, that’s where mongoose will come into play. It would start triggering some validations over the properties inside of this posts right here. Now joe will behave exactly same as it was bahaveing previously. Then we save joe. Code-

Subdocument\_test.js-

const assert = require('assert');

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

describe('Subdocuments', () => {

it('can create a subdcument', (done) => {

const joe = new User({

name: 'Joe',

posts: [{title: 'PostTitle'}]

});

joe.save()

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

.then(user => {

assert(user.posts[0].title === 'PostTitle');

done();

});

});

});

So what do we really gain by making this clear association between user and lists of posts? We can add any property to user. So why did all that? In next lecture w e will see that how we can use this subdocument idea to get couple of better proeprties or bettr features out of mongoose.

48)Adding Subdocuments to Existing records

Code-

it('Can add subdocuments to an existing recird', (done) => {

const joe = new User({

name: 'Joe',

posts: []

});

joe.save()

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

.then(user => {

user.posts.push({title: 'New Post'});

return user.save();

})

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

.then(user => {

assert(user.posts[0].title === 'New Post');

done();

});

});

Note here we provided empty posts property to joe, before saving it in db. it is not mandatory. Same code will work fine if you do not even give posts property to user before saving.

49)Removing sub documents

Here we want to remove a posts from user document. So first we save document. Now you can say like we used push method for inserting, we can use splice for removal. Well we definitely can but mongoose recognozes the use of splice is kind of pain in the rear.lets imagine for a second that we had a bunch of different records inside of this posts subdocument collection. We would have to kind of walk through that list and find a very particular record that we wanted. We would have to kind of like walk through that list and find a very particular record that we wanted to pull out. Ad then you know it’s such a pain in the rear, I don’t know if you have done this with javascript but you have got to find the index of that record you want to remove and then call splice that with index. In sort its kind of pain using raw js to remove elements out of an array. So mongoose gives us a little bit of alternative api for removing a record out of a nested collection of records. Code-

We have removed post.but we need to save user back to db to make changes on user. Remember we used remove on user then we do not need to call save. This is because joe is collection. It makes a actual operation on database. Whenw e remove subdocument, same is not true. It will not save our record back to mongo. It just removes it out of subdocument collection of posts and user is still kind of dangling, like my ist of posts changed but I guess I don’t really have to do anything about that. So after we remove a sub document out of nested collection like this we still have to manually call save on parent record whci is user for us.

50)Virtual types

This is our user 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,

posts: [PostSchema]

});

Here we have posts and postCount. Now postCount are not linked with posts, we have to manually keep track of postCount.we want this-postCount should look inside posts and returns however many posts are inside this posts array.

For this we are going to use idea of virtual types inside of mongoose . a virtual type is any field on a model(like postCount in our case) that does not actually get persisited over to our mongo db. so any time we try to access to postCount property, we will calculate length of posts array and return that number.

So virtual type or virtual property or virtual field refers to any type of property that we have on our model that does not actually get saved over to our mongo db. whenever you are thinking about a property on a model that is like derivative or like product of 2 or more properties, a fantastic solution is to start thinking about virtual types.

Create a new file virtual\_type\_test.js in test folder. here first we will write test first then we will write code-

Virtual\_type\_test.js-

const assert = require('assert');

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

describe('Virtual Types', () => {

it('postCount returns number of posts', (done) => {

const joe = new User({

name: 'Joe',

posts: [{title: 'PostTitle'}]

});

joe.save()

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

.then(user => {

assert(joe.postCount === 1);

done();

});

});

});

We can test how this postCount thing works without saving joe to database. Here I want to give you a realistic example.

51)Defining a virtual Type

Here we will change our model to define virtual property. Right now we define our user model like this-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = require('./post');

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,

posts: [PostSchema]

});

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

module.exports = User;

now we want postCount to be virtual field. We do not want it to be saved into database. We want it to be calculated on the fly. So nay time we are using virtual property or something that should be calculated on the fly, we will not define this property inside of schema. So we remove it from schema. But we are using this postCount property in update\_test.js. change it to xit. In this way, that test won’t run-

xit("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();

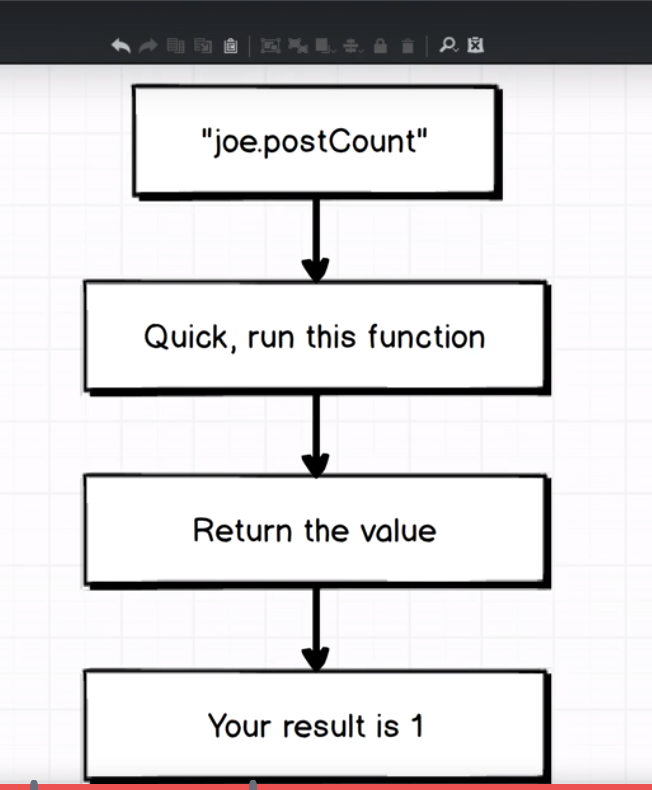
});

});

Now lets go back to define virtual property. We define virtual property outside user schema-

UserSchema.virtual('postCount')

Now we hve to tell what to do whenever someone access postCount property. To set function on this thing or kind of define this virtual property. We chain on get and we pass function to get. This function should not be fat arrow function. lets seediagram to see what we are doing-



These virtual property work by using getter and setter of ES6. Whenever we define getter with virtual property, then we can refrence a property on our userModel like joe.postCount.

When we access joe.postCount something is going to happen which is very different from the type of javascript that we are used to. In js we used to, joe.postCount is going to return postCount property on joe. But here when we try to access joe.postCount, mongo is going to run getter function and it will return value that is returned by getter function.

UserSchema.virtual('postCount').get(function() {

});

52)ES6 Getters

Here we saw that inside getter function, this points to whole document on which we accessed postCount. This why we have not used fat arrow function. we use arrow function inside of get, then context or value of this is going to be bound to this whole file(user.js). so this will not refer to instance of user model, it will be what this is outside this function9i.e this that is in global scope in that file). so we define our model like this-

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const PostSchema = require('./post');

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']

},

posts: [PostSchema]

});

UserSchema.virtual('postCount').get(function() {

return this.posts.length;

});

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

module.exports = User;

now run your tests. We will see that one test is pending. This is because we have xit in place of it.

53)Fixing update tests

Here we defined a new property called likes on user schema. So didi so that we can make our update test that uses inc work again. Then we changed postCount with likes in update\_file.