**Associations**

Using Mongoose, we are able to create associations between Mongo objects.

**How does it work?**

Basically, all we will do is associate two different Mongoose models together using a bit of additional syntax.  In this example, we're going to show you how to affiliate a post model with a comment model. This is a   **one-to-many relationship**( 1 Post, to Many Comments ).

Note: One-to-one relationships are easily accomplished with a small change in syntax.

**Setup**

Before you define your models, you must establish a  *Schema* variable like the one below. You'll be calling this variable inside of all the models that require the associations. Why? Well, this will enable your models to read and understand those huge *ObjectId*attributes that MongoDB automatically generated!

**var Schema = mongoose.Schema;**

**Post Model**

var Schema = mongoose.Schema;

var PostSchema = new mongoose.Schema({

text: { type: String, required: true },

comments: [{type: Schema.Types.ObjectId, ref: 'Comment'}]

}, { timestamps: true });

*// The 'type' property of the object inside of the array is an attribute*

*// that tells Mongoose what to look for.*

Do you see what is going on here? First of all, we know this is a   **one-to-many relationship** because the **comments property is an array**, meaning the **post model** will have many comments. Secondly, the object we are going to have many of has two properties:**type**and**ref.** The **type property** tells Mongoose the**unique identifier**for our association, which for us is the **ObjectID** of each **comment.** The**ref**property tells Mongoose what model the unique identifier is associated with.  Long story short:

#### Mongoose is just keeping an array full of associated comment IDs

Which is, if you think about it, the only way we could accomplish this with MongoDB as it is.

### Comment Model

*// What would we need to add to make the below snippet work properly? Read your console!*

var CommentSchema = new mongoose.Schema({

*// since this is a reference to a different document, the \_ is the naming convention!*

\_post: {type: Schema.Types.ObjectId, ref: 'Post'},

text: { type: String, required: true },

}, {timestamps: true });

**Now we must tell the comment model it belongs to a post model**. When we are referencing a model that another model belongs to, **we use an underscore ( \_ )** to indicate this is another model. Note the small variation in syntax in the type property.

### Retrieving the data

We're going to query our data a little bit differently than we have before, but the changes aren't huge. The important methods here are the   **.populate() method**and **the .exec() method**. The **.populate()**method basically tells your post model to gather the associated comments and the **.exec()** method actually fires the action.

### Here is how to get the comments for one post:

*// just an example route, your routes may look different*

app.get('/posts/:id', function (req, res){

*// the populate method is what grabs all of the comments using their IDs stored in the*

*// comment property array of the post document!*

Post.findOne({\_id: req.params.id})

.populate('comments')

.exec(function(err, post) {

res.render('post', {post: post});

});

});

### Adding a comment

If you've done anything with Rails, you know that for an  activeRecord association, as long as you update one end of the relationship, (post or comment in this case) the other end will be updated automatically. Sadly, Mongoose is not as thorough. In order to add a comment to a preexisting post, we must update both parties. The code to do that is as follows:

*// just a sample route. Post request to update a post.*

*// your routes will probably look different.*

app.post('/posts/:id', function (req, res){

Post.findOne({\_id: req.params.id}, function(err, post){

*// data from form on the front end*

var comment = new Comment(req.body);

*// set the reference like this:*

comment.\_post = post.\_id;

*// now save both to the DB*

comment.save(function(err){

post.comments.push(comment);

post.save(function(err){

if(err) {

console.log('Error');

} else {

res.redirect('/');

}

});

});

});

});

This process is straightforward, but a little tedious. This functionality is useful for basic one-to-many or one-to-one relationships. Obviously, the syntax would change for a one-to-one relationship, but the execution is still the same. Keep in mind this process could get pretty messy for a many-to-many relationship, so avoid building one if at all possible in your app. That being said, if you are building an application that uses a many-to-many relationship,  **you probably shouldn't be using MongoDB in the first place!**

Here it is all together.

*// define Schema variable*

var Schema = mongoose.Schema;

*// define Post Schema*

var PostSchema = new mongoose.Schema({

text: {type: String, required: true },

comments: [{type: Schema.Types.ObjectId, ref: 'Comment'}]

}, {timestamps: true });

*// define Comment Schema*

var CommentSchema = new mongoose.Schema({

\_post: {type: Schema.Types.ObjectId, ref: 'Post'},

text: {type: String, required: true }

}, {timestamps: true });

*// set our models by passing them their respective Schemas*

mongoose.model('Post', PostSchema);

mongoose.model('Comment', CommentSchema);

*// store our models in variables*

var Post = mongoose.model('Post');

var Comment = mongoose.model('Comment');

*// route for getting a particular post and comments*

app.get('/posts/:id', function (req, res){

Post.findOne({\_id: req.params.id})

.populate('comments')

.exec(function(err, post) {

res.render('post', {post: post});

});

});

*// route for creating one comment with the parent post id*

app.post('/posts/:id', function (req, res){

Post.findOne({\_id: req.params.id}, function(err, post){

var comment = new Comment(req.body);

comment.\_post = post.\_id;

post.comments.push(comment);

comment.save(function(err){

post.save(function(err){

if(err) { console.log('Error'); }

else { res.redirect('/'); }

});

});

});

});

### ****Embedded Documents****

Let's say you find yourself  **always retrieving associative documents in groups, and never by themselves**. Luckily for you, there is a better way! Consider the example below:

{

\_id: "dojo",

name: "CodingDojo"

}

{

business\_id: "dojo",

street: "10777 Main Street",

city: "Bellevue",

state: "WA",

zip: "98004"

}

{

business\_id: "dojo",

street: "1980 Zanker Road",

city: "San Jose",

state: "CA",

zip: "95112"

}

Hmm. You can probably tell that the above example will take a ton of queries to properly retrieve. First, you need to get the business document with the  **\_id attribute of 'dojo'**, then you need to hunt around for **every location with the business\_id of 'dojo'**! That's double the queries! Let's say that in your web application, you're never pulling businesses by themselves, but always with locations at the same time. That's a lot of work! Consider the below example:

**{**

**name: "CodingDojo",**

**locations: [**

**{**

**street: "10777 Main Street",**

**city: "Bellevue",**

**state: "WA",**

**zip: "98004"**

**},**

**{**

**street: "1980 Zanker Road",**

**city: "San Jose",**

**state: "CA",**

**zip: "95112"**

**}**

**]**

**}**

Notice that we took those multiple documents and turned them into one!  **This is called embedding documents.** If you embed the locations within your business document, **you can get all of your addresses out all in the same query.** Always put thought into how you structure your data! Making choices on how you deal with associative data at the beginning of a project can save you tons of headaches later in development.