# Simple example - Node.js, Restify, MongoDb and Mongoose

Before I start, the Backbone.js parts of this tutorial will be using techniques described in "Organizing your application using [Modules](http://backbonetutorials.com/organizing-backbone-using-modules/) to construct a simple guestbook.

## Getting started

To easily understand this tutorial you should jump straight into the example code base.

[Example Codebase](https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/nodejs-mongodb-mongoose-restify)

[Example Demo](https://thomasdavis.github.io/backbonetutorials/examples/nodejs-mongodb-mongoose-restify/app)

This tutorial will assist you in saving data(Backbone.js Models) to MongoDb and retrieving a list(Backbone.js Collections) of them back.

## The technologies

This stack is great for rapid prototyping and highly intuitive. Personal note: I love using JavaScript as my only language for the entire application (FrontEnd/BackEnd/API/Database). Restify is still in early development but is essentially just an extension of Express. So for anyone needing more stability you can easily just substitute Express in.

### Node.js

"Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices."

### Restify

"Restify is a node.js module built specifically to enable you to build correct REST web services. It borrows heavily from express (intentionally) as that is more or less the de facto API for writing web applications on top of node.js."

### MongoDb

"MongoDB (from "humongous") is a scalable, high-performance, open source NoSQL database."

### Mongoose

"Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment."

## Building the server

In the example repository there is a server.js example which can be executed by running node server.js. If you use this example in your own applications make sure to update the Backbone.js [Model](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/models/message.js) and [Collection](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/collections/messages.js) definitions to match your server address.

## Restify configuration

The first thing to do is require the Restify module. Restify will be in control of handling our restful endpoints and returning the appropriate JSON.

**var** restify = require('restify');

**var** server = restify.createServer();

server.use(restify.bodyParser());

Note: bodyParser() takes care of turning your request data into a JavaScript object on the server automatically.

## MongoDb/Mongoose configuration

We simply want to require the MongoDb module and pass it a MongoDb authentication URI e.g. mongodb://username:server@mongoserver:10059/somecollection

The code below presupposes you have another file in the same directory called config.js. Your config should never be public as it contains your credentials. So for this repository I have added config.js to my .gitignore but added in a [sample config](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/config-sample.js).

**var** mongoose = require('mongoose/');

**var** config = require('./config');

db = mongoose.connect(config.creds.mongoose\_auth),

Schema = mongoose.Schema;

## Mongoose Schema

Mongoose introduces a concept of [model/schema](http://mongoosejs.com/docs/model-definition.html) enforcing types which allow for easier input validation etc

// Create a schema for our data

**var** MessageSchema = **new** Schema({

message: String,

date: Date

});

// Use the schema to register a model with MongoDb

mongoose.model('Message', MessageSchema);

**var** Message = mongoose.model('Message');

\_Note: Message can now be used for all things CRUD related.

## Setting up the routes

Just like in Backbone, Restify allows you to configure different routes and their associated callbacks. In the code below we define two routes. One for saving new messages and one for retrieving all messages. After we have created our function definitions, we attach them to either GET/POST/PUT/DELETE on a particular restful endpoint e.g. GET /messages

// This function is responsible for returning all entries for the Message model

**function** **getMessages**(req, res, next) {

// Resitify currently has a bug which doesn't allow you to set default headers

// This headers comply with CORS and allow us to server our response to any origin

res.header("Access-Control-Allow-Origin", "\*");

res.header("Access-Control-Allow-Headers", "X-Requested-With");

// .find() without any arguments, will return all results

// the `-1` in .sort() means descending order

Message.find().sort('date', -1).execFind(**function** (arr,data) {

res.send(data);

});

}

**function** **postMessage**(req, res, next) {

res.header("Access-Control-Allow-Origin", "\*");

res.header("Access-Control-Allow-Headers", "X-Requested-With");

// Create a new message model, fill it up and save it to Mongodb

**var** message = **new** Message();

message.message = req.params.message;

message.date = **new** Date();

message.save(**function** () {

res.send(req.body);

});

}

// Set up our routes and start the server

server.get('/messages', getMessages);

server.post('/messages', postMessage);

This wraps up the server side of things, if you follow the [example](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/server.js) then you should see something like

<http://backbonetutorials.nodejitsu.com/messages>

Note: Again you must remember to change the[*Model*](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/models/message.js)and[*Collection*](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/js/collections/messages.js)definitions to match your server address.

## Setting up the client (Backbone.js)

I've actually used the latest copy of [http://backboneboilerplate.com](http://backboneboilerplate.com/) to set up the example page.

The important files you will want to check out are;

* views/dashboard/page.js
* views/guestbook/form.js
* views/guestbook/list.js
* models/message.js
* collections/messages.js
* templates/guestbook/

## Saving a message

First of all we want to setup a [template](https://github.com/thomasdavis/backbonetutorials/blob/gh-pages/examples/nodejs-mongodb-mongoose-restify/templates/guestbook/form.html) for showing our form that creates new messages.

<**textarea** class="message"></**textarea**>

<**button** class="post-message">Post Message</**button**>

This template gets inserted into the DOM by views/guestbook/form.js, this Backbone view also handles the interaction of the form and the posting of the new data.

Let us create a Backbone Model that has the correct URL for our restful interface.

define([

'underscore',

'backbone'

], **function**(\_, Backbone) {

**var** Message = Backbone.Model.extend({

url: 'http://localhost:8080/messages'

});

**return** Message;

});

We can see how we require our predefined model for messages and also our form template.

define([

'jquery',

'underscore',

'backbone',

'models/message',

'text!templates/guestbook/form.html'

], **function**($, \_, Backbone, MessageModel, guestbookFormTemplate){

**var** GuestbookForm = Backbone.View.extend({

el: '.guestbook-form-container',

render: **function** () {

$(**this**.el).html(guestbookFormTemplate);

},

events: {

'click .post-message': 'postMessage'

},

postMessage: **function**() {

**var** that = **this**;

**var** message = **new** MessageModel();

message.save({ message: $('.message').val()}, {

success: **function** () {

that.trigger('postMessage');

}

});

}

});

**return** GuestbookForm;

});

Note: trigger is from Backbone Events, I binded a listener to this view in views/dashboard/page.js so when a new message is submitted, the list is re-rendered. We are setting the date of the POST on the server so there is no need to pass it up.

## Retrieving a list of messages

We setup a route on our server to generate a list of all available messages at GET /messages. So we need to define a collection with the appropriate url to fetch this data down.

define([

'jquery',

'underscore',

'backbone',

'models/message'

], **function**($, \_, Backbone, MessageModel){

**var** Messages = Backbone.Collection.extend({

model: MessageModel, // Generally best practise to bring down a Model/Schema for your collection

url: 'http://localhost:8080/messages'

});

**return** Messages;

});

Now that we have a collection to use we can setup our views/list.js to require the collection and trigger a fetch. Once the fetch is complete we want to render our returned data to a template and insert it into the DOM.

define([

'jquery',

'underscore',

'backbone',

'collections/messages',

'text!templates/guestbook/list.html'

], **function**($, \_, Backbone, MessagesCollection, guestbookListTemplate){

**var** GuestbookList = Backbone.View.extend({

el: '.guestbook-list-container',

render: **function** () {

**var** that = **this**;

**var** messages = **new** MessagesCollection();

messages.fetch({

success: **function**(messages) {

$(that.el).html(\_.template(guestbookListTemplate, {messages: messages.models, \_:\_}));

}

});

}

});

**return** GuestbookList;

});

The template file should iterate over messages.models which is an array and print out a HTML fragment for each model.

<**%** \_.each(messages, function(message) { %>

<**p**><**%=** message.get('message') %></**p**>

<**em**><**%=** message.get('date') %></**em**>

<**%** }); %>

This actually sums up everything you need to know to implement this simple example.

## Conclusion

[Example Codebase](https://github.com/thomasdavis/backbonetutorials/tree/gh-pages/examples/nodejs-mongodb-mongoose-restify)

[Example Demo](https://thomasdavis.github.io/backbonetutorials/examples/nodejs-mongodb-mongoose-restify/app)

In this example you should really be using relative URL's in your collections/models and instead setting a baseUrl in a config file or by placing your index.html file on the restful server.

This example is hosted on GitHub therefore we had to include the absolute URL to the server which is hosted on nodejitsu.com

On a personal note, I have of recent used the Joyent, Nodejitsu, MongoDbHq stack after they have now partnered up and I have nothing but good things to say. Highly recommend you check it out!

As always I hope I made this tutorial easy to follow!

Get in touch with me on twitter, comments or GitHub!

### Relevant Links

[Organizing Your Backbone.js Application With Modules](http://weblog.bocoup.com/organizing-your-backbone-js-application-with-modules)

[Storing Data With Node.js And Mongoose](https://www.kompulsa.com/introduction-mongoose-storing-data-mongodb/)

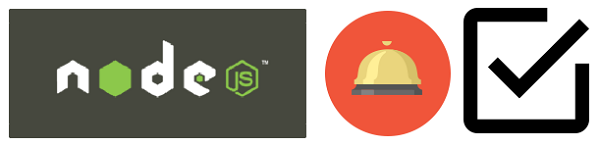
### Author

RESTful API Design With NodeJS & Restify

by [Hüseyin Babal](https://tutsplus.com/authors/huseyin-babal)9 Dec 2014

Difficulty:IntermediateLength:LongLanguages:

[Node.js](https://code.tutsplus.com/categories/nodejs)[REST API](https://code.tutsplus.com/categories/rest-api)[Designing](https://code.tutsplus.com/categories/designing)[JavaScript](https://code.tutsplus.com/categories/javascript)[Express](https://code.tutsplus.com/categories/express)[Web Apps](https://code.tutsplus.com/categories/web-apps)

What You'll Be Creating

The RESTful API consists of two main concepts: **Resource**, and **Representation**. Resource can be any object associated with data, or identified with a URI (more than one URI can refer to the same resource), and can be operated using HTTP methods. Representation is the way you display the resource. In this tutorial we will cover some theoretical information about RESTful API design, and implement an example blogging application API by using NodeJS.

**Resource**

Choosing the correct resources for a RESTful API is an important section of designing. First of all, you need to analyze your business domain and then decide how many and what kind of resources will be used that are relevant to your business need. If you are designing a blogging API, you will probably use **Article**, **User**, and **Comment**. Those are the resource names, and the data associated with that is the resource itself:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11 | {      "title": "How to Design RESTful API",      "content": "RESTful API design is a very important case in the software development world.",      "author": "huseyinbabal",      "tags": [          "technology",          "nodejs",          "node-restify"          ]      "category": "NodeJS"  } |

**Resource Verbs**

You can proceed with a resource operation after you have decided on the required resources. Operation here refers to HTTP methods. For example, in order to create an article, you can make the following request:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | POST /articles HTTP/1.1  Host: localhost:3000  Content-Type: application/json    {    "title": "RESTful API Design with Restify",    "slug": "restful-api-design-with-restify",    "content": "Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.",    "author": "huseyinbabal"  } |

In the same way, you can view an existing article by issuing the following request:

|  |  |
| --- | --- |
| 1  2  3 | GET /articles/123456789012 HTTP/1.1  Host: localhost:3000  Content-Type: application/json |

What about updating an existing article? I can hear that you are saying:

*I can make another POST request to /articles/update/123456789012 with the payload.*

Maybe preferable, but the URI is becoming more complex. As we said earlier, operations can refer to HTTP methods. This means, state the **update**operation in the HTTP method instead of putting that in the URI. For example:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | PUT /articles/123456789012 HTTP/1.1  Host: localhost:3000  Content-Type: application/json  {      "title": "Updated How to Design RESTful API",      "content": "Updated RESTful API design is a very important case in the software development world.",      "author": "huseyinbabal",      "tags": [          "technology",          "nodejs",          "restify",          "one more tag"          ]      "category": "NodeJS"  } |

By the way, in this example you see tags and category fields. Those don't need to be mandatory fields. You can leave them blank and set them in future.

Sometimes, you need to delete an article when it is outdated. In that case you can use a **DELETE**HTTP request to **/articles/123456789012.**

HTTP methods are standard concepts. If you use them as an operation, you will have simple URIs, and this kind of simple API will help you gain happy consumers.

What if you want to insert a comment to an article? You can select the article and add a new comment to the selected article. By using this statement, you can use the following request:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | POST /articles/123456789012/comments HTTP/1.1  Host: localhost:3000  Content-Type: application/json  {      "text": "Wow! this is a good tutorial",      "author": "john doe"  } |

The above form of resource is called as a **sub-resource. Comment**is a sub-resource of **Article.**The **Comment**payload above will be inserted in the database as a child of **Article**. Sometimes, a different URI refers to the same resource. For example, to view a specific comment, you can use either:

|  |  |
| --- | --- |
| 1  2  3 | GET /articles/123456789012/comments/123 HTTP/1.1  Host: localhost:3000  Content-Type: application/json |

or:

|  |  |
| --- | --- |
| 1  2  3 | GET /comments/123456789012 HTTP/1.1  Host: localhost:3000  Content-Type: application/json |

**Versioning**

In general, API features change frequently in order to provide new features to consumers. In that case, two versions of the same API can exist at the same time. In order to separate those two features, you can use versioning. There are two forms of versioning

1. **Version in URI:**You can provide the version number in the URI. For example, **/v1.1/articles/123456789012**.
2. **Version in Header:**Provide the version number in the header, and never change the URI.For example:

|  |  |
| --- | --- |
| 1  2  3 | GET /articles/123456789012 HTTP/1.1  Host: localhost:3000  Accept-Version: 1.0 |

Actually, the version changes only the representation of the resource, not the concept of the resource. So, you do not need to change the URI structure. In v1.1, maybe a new field was added to Article. However, it still returns an article. In the second option, the URI is still simple and consumers do not need to change their URI in client-side implementations.

It is important to design a strategy for situations where the consumer does not provide a version number. You can raise an error when version is not provided, or you can return a response by using the first version. If you use the latest stable version as a default, consumers can get many errors for their client-side implementations.

**Representation**

Representation is the way that an API displays the resource. When you call an API endpoint, you will get returned a resource. This resource can be in any format like XML, JSON, etc. JSON is preferable if you are designing a new API. However, if you are updating an existing API that used to return an XML response, you can provide another version for a JSON response.

That's enough theoretical information about RESTful API design. Let's have a look at real life usage by designing and implementing a Blogging API using Restify.

**Blogging REST API**

**Design**

In order to design a RESTful API, we need to analyze the business domain. Then we can define our resources. In a Blogging API, we need:

* Create, Update, Delete, View **Article**
* Create a comment for a specific **Article**, Update, Delete, View, **Comment**
* Create, Update, Delete, View **User**

In this API, I will not cover how to authenticate a user in order to create an article or comment. For the authentication part, you can refer to the [Token-Based Authentication with AngularJS & NodeJS](http://code.tutsplus.com/tutorials/token-based-authentication-with-angularjs-nodejs--cms-22543) tutorial.

Our resource names are ready. Resource operations are simply CRUD. You can refer to the following table for a general showcase of API.

| **Resource Name** | **HTTP Verbs** | **HTTP Methods** |
| --- | --- | --- |
| Article | create Article update Article delete Article view Article | POST /articles with Payload PUT /articles/123 with Payload DELETE /articles/123 GET /article/123 |
| Comment | create Comment update Coment delete Comment view Comment | POST /articles/123/comments with Payload PUT /comments/123 with Payload DELETE /comments/123 GET /comments/123 |
| User | create User update User delete User view User | POST /users with Payload PUT /users/123 with Payload DELETE /users/123 GET /users/123 |

Advertisement

**Project Setup**

In this project we will use **NodeJS** with **Restify**. The resources will be saved in the **MongoDB** database. First of all, we can define resources as models in Restify.

**Article**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13 | var mongoose = require("mongoose");  var Schema   = mongoose.Schema;    var ArticleSchema = new Schema({      title: String,      slug: String,      content: String,      author: {          type: String,          ref: "User"      }  });  mongoose.model('Article', ArticleSchema); |

**Comment**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15 | var mongoose = require("mongoose");  var Schema   = mongoose.Schema;    var CommentSchema = new Schema({      text: String,      article: {          type: String,          ref: "Article"      },      author: {          type: String,          ref: "User"      }  });  mongoose.model('Comment', CommentSchema); |

**User**

There won't be any operation for the User resource. We will assume that we already know the current user who will be able to operate on articles or comments.

You may ask where this [mongoose](http://mongoosejs.com/index.html) module comes from. It is the most popular ORM framework for MongoDB written as a NodeJS module. This module is included in the project within another config file.

Now we can define our HTTP verbs for the above resources. You can see the following:

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | var restify = require('restify')      , fs = require('fs')      var controllers = {}      , controllers\_path = process.cwd() + '/app/controllers'  fs.readdirSync(controllers\_path).forEach(function (file) {      if (file.indexOf('.js') != -1) {          controllers[file.split('.')[0]] = require(controllers\_path + '/' + file)      }  })    var server = restify.createServer();    server      .use(restify.fullResponse())      .use(restify.bodyParser())    // Article Start  server.post("/articles", controllers.article.createArticle)  server.put("/articles/:id", controllers.article.updateArticle)  server.del("/articles/:id", controllers.article.deleteArticle)  server.get({path: "/articles/:id", version: "1.0.0"}, controllers.article.viewArticle)  server.get({path: "/articles/:id", version: "2.0.0"}, controllers.article.viewArticle\_v2)  // Article End    // Comment Start  server.post("/comments", controllers.comment.createComment)  server.put("/comments/:id", controllers.comment.viewComment)  server.del("/comments/:id", controllers.comment.deleteComment)  server.get("/comments/:id", controllers.comment.viewComment)  // Comment End    var port = process.env.PORT || 3000;  server.listen(port, function (err) {      if (err)          console.error(err)      else          console.log('App is ready at : ' + port)  })    if (process.env.environment == 'production')      process.on('uncaughtException', function (err) {          console.error(JSON.parse(JSON.stringify(err, ['stack', 'message', 'inner'], 2)))      }) |

In this code snippet, first of all the controller files that contain controller methods are iterated and all the controllers are initialized in order to execute a specific request to the URI. After that, URIs for specific operations are defined for basic CRUD operations. There is also versioning for one of the operations on Article.

For example, if you state version as 2 in Accept-Version header, viewArticle\_v2 will be executed. viewArticle and viewArticle\_v2 both do the same job, showing the resource, but they show Article resource in a different format, as you can see in the title field below. Finally, the server is started on a specific port, and some error reporting checks are applied. We can proceed with controller methods for HTTP operations on resources.

**article.js**

|  |  |
| --- | --- |
| 001  002  003  004  005  006  007  008  009  010  011  012  013  014  015  016  017  018  019  020  021  022  023  024  025  026  027  028  029  030  031  032  033  034  035  036  037  038  039  040  041  042  043  044  045  046  047  048  049  050  051  052  053  054  055  056  057  058  059  060  061  062  063  064  065  066  067  068  069  070  071  072  073  074  075  076  077  078  079  080  081  082  083  084  085  086  087  088  089  090  091  092  093  094  095  096  097  098  099  100  101  102  103  104  105  106  107  108  109  110  111  112 | var mongoose = require('mongoose'),      Article = mongoose.model("Article"),      ObjectId = mongoose.Types.ObjectId    exports.createArticle = function(req, res, next) {      var articleModel = new Article(req.body);      articleModel.save(function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              res.json({                  type: true,                  data: article              })          }      })  }    exports.viewArticle = function(req, res, next) {      Article.findById(new ObjectId(req.params.id), function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              if (article) {                  res.json({                      type: true,                      data: article                  })              } else {                  res.json({                      type: false,                      data: "Article: " + req.params.id + " not found"                  })              }          }      })  }    exports.viewArticle\_v2 = function(req, res, next) {      Article.findById(new ObjectId(req.params.id), function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              if (article) {                  article.title = article.title + " v2"                  res.json({                      type: true,                      data: article                  })              } else {                  res.json({                      type: false,                      data: "Article: " + req.params.id + " not found"                  })              }          }      })  }    exports.updateArticle = function(req, res, next) {      var updatedArticleModel = new Article(req.body);      Article.findByIdAndUpdate(new ObjectId(req.params.id), updatedArticleModel, function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              if (article) {                  res.json({                      type: true,                      data: article                  })              } else {                  res.json({                      type: false,                      data: "Article: " + req.params.id + " not found"                  })              }          }      })  }    exports.deleteArticle = function(req, res, next) {      Article.findByIdAndRemove(new Object(req.params.id), function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              res.json({                  type: true,                  data: "Article: " + req.params.id + " deleted successfully"              })          }      })  } |

You can find an explanation of basic CRUD operations on the Mongoose side below:

* **createArticle:**This is a simple **save**operation on articleModel sent from the request body. A new model can be created by passing the request body as a constructor to a model like var articleModel = new Article(req.body).
* **viewArticle:**In order to view article detail, an article ID is needed in the URL parameter. findOne with an ID parameter is enough to return article detail.
* **updateArticle:**Article update is a simple find query and some data manipulation on the returned article. Finally, the updated model needs to be saved to the database by issuing a save command.
* **deleteArticle:**findByIdAndRemove is the best way to delete an article by providing the article ID.

The Mongoose commands mentioned above are simply static like method through Article object that is also a reference of the Mongoose schema.

**comment.js**

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | var mongoose = require('mongoose'),      Comment = mongoose.model("Comment"),      Article = mongoose.model("Article"),      ObjectId = mongoose.Types.ObjectId    exports.viewComment = function(req, res) {      Article.findOne({"comments.\_id": new ObjectId(req.params.id)}, {"comments.$": 1}, function(err, comment) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              if (comment) {                  res.json({                      type: true,                      data: new Comment(comment.comments[0])                  })              } else {                  res.json({                      type: false,                      data: "Comment: " + req.params.id + " not found"                  })              }          }      })  }    exports.updateComment = function(req, res, next) {      var updatedCommentModel = new Comment(req.body);      console.log(updatedCommentModel)      Article.update(          {"comments.\_id": new ObjectId(req.params.id)},          {"$set": {"comments.$.text": updatedCommentModel.text, "comments.$.author": updatedCommentModel.author}},          function(err) {              if (err) {                  res.status(500);                  res.json({                      type: false,                      data: "Error occured: " + err                  })              } else {                  res.json({                      type: true,                      data: "Comment: " + req.params.id + " updated"                  })              }      })  }    exports.deleteComment = function(req, res, next) {      Article.findOneAndUpdate({"comments.\_id": new ObjectId(req.params.id)},          {"$pull": {"comments": {"\_id": new ObjectId(req.params.id)}}},          function(err, article) {          if (err) {              res.status(500);              res.json({                  type: false,                  data: "Error occured: " + err              })          } else {              if (article) {                  res.json({                      type: true,                      data: article                  })              } else {                  res.json({                      type: false,                      data: "Comment: " + req.params.id + " not found"                  })              }          }      })  } |

When you make a request to one of the resource URIs, the related function stated in the controller will be executed. Every function inside the controller files can use the **req** and **res** objects. The **comment** resource here is a sub-resource of **Article.**All the query operations are made through the Article model in order to find a sub-document and make the necessary update. However, whenever you try to view a Comment resource, you will see one even if there is no collection in MongoDB.

**Other Design Suggestions**

* Select easy-to-understand resources in order to provide easy usage to consumers.
* Let business logic be implemented by consumers. For example, the Article resource has a field called **slug.**Consumers do not need to send this detail to the REST API. This slug strategy should manage on the REST API side to reduce coupling between API and consumers. Consumers only need to send title detail, and you can generate the slug according to your business needs on the REST API side.
* Implement an authorization layer for your API endpoints. Unauthorized consumers can access restricted data that belongs to another user. In this tutorial, we did not cover the User resource, but you can refer to [Token Based Authentication with AngularJS & NodeJS](http://code.tutsplus.com/tutorials/token-based-authentication-with-angularjs-nodejs--cms-22543) for more information about API authentications.
* User URI instead of query string. /articles/123  (Good), /articles?id=123 (Bad).
* Do not keep the state; always use instant input/output.
* Use noun for your resources. You can use HTTP methods in order to operate on resources.

Finally, if you design a RESTful API by following these fundamental rules, you will always have a flexible, maintainable, easily understandable system.