Building your first application with MongoDB: Creating a REST API using the MEAN Stack - Part 2

Register for an online course (https://university.mongodb.com/?jmp=hero)

< View all blog posts (/blog)

Norberto Leite (/blog?author=555f57de41636850f3050500)
April 16, 2015

[illegible]

In the first part of this blog series (https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1), we covered the basic mechanics of our application and undertook some data modeling. In this second part, we will create tests that validate the behavior of our application and then describe how to set-up and run the application.

Write the tests first

Let’s begin by defining some small configuration libraries.

file name: **test/config/test_config.js**

```
module.exports = {
    url : 'http://localhost:8000/api/v1.0'
}
```

terzawistowicz/7ce95f70da252a5e6bbc/raw/9acd971e8ccee566045774a486b70b58cc196ae5/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/7ce95f70da252a5e6bbc#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Our server will be running on port 8000 on localhost. This will be fine for initial testing purposes. Later, if we change the location or port number for a production system, it would be very easy to just edit this file.

To prepare for our test cases, we need to ensure that we have a good test environment. The following code achieves this for us. First, we connect to the database.

file name: **test/setup_tests.js**

```
function connectDB(callback) {
    mongoClient.connect(dbConfig.testDBURL, function(err, db) {
        assert.equal(null, err);
        reader_test_db = db;
        console.log("Connected correctly to server");
        callback(0);
    });
}
```

eterzawistowicz/92593e2ee273d6ad96f8/raw/c3517655667505b3defaa2428535c46a9b72a560/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/92593e2ee273d6ad96f8#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we drop the user collection. This ensures that our database is in a known starting state.

```
function dropUserCollection(callback) {
        console.log("dropUserCollection");
        user = reader_test_db.collection('user');
        if (undefined != user) {
            user.drop(function(err, reply) {
                console.log('user collection dropped');
                callback(0);
            });
        } else {
            callback(0);
        }
    },
```

peterzawistowicz/528087bd48c386fdf00e/raw/db32b8722bb438f241be2b05790cfc48717fc824/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/528087bd48c386fdf00e#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we will drop the user feed entry collection.

```
    function dropUserFeedEntryCollection(callback) {
        console.log("dropUserFeedEntryCollection");
        user_feed_entry = reader_test_db.collection('user_feed_entry');
        if (undefined != user_feed_entry) {
            user_feed_entry.drop(function(err, reply) {
                console.log('user_feed_entry collection dropped');
                callback(0);
            });
        } else {
            callback(0);
        }
    }
```

terzawistowicz/e81a9c0e45bcb779d10d/raw/09b1fe932190e45b56d585da2e3041c665c61398/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/e81a9c0e45bcb779d10d#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we will connect to Stormpath and delete all the users in our test application.

```
function getApplication(callback) {
        console.log("getApplication");
        client.getApplications({
            name: SP_APP_NAME
        }, function(err, applications) {
            console.log(applications);
            if (err) {
                log("Error in getApplications");
                throw err;
            }
            app = applications.items[0];
            callback(0);
        });
    },
    function deleteTestAccounts(callback) {
        app.getAccounts({
            email: TU_EMAIL_REGEX
        }, function(err, accounts) {
            if (err) throw err;
            accounts.items.forEach(function deleteAccount(account) {
                account.delete(function deleteError(err) {
                    if (err) throw err;
                });
            });
            callback(0);
        });
    }
```

peterzawistowicz/426300a0bce5e3389cc3/raw/779a9d73ef9252e7de8af6c9f8499b123ff10ed9/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/426300a0bce5e3389cc3#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we close the database.

```
function closeDB(callback) {
    reader_test_db.close();
}
```

peterzawistowicz/b6a428fef9d96d8844aa/raw/0e4de1d4ceaeb695c8dab5ff229a35b8c4715ebd/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/b6a428fef9d96d8844aa#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Finally, we call async.series to ensure that all the functions run in the correct order.

```
async.series([connectDB, dropUserCollection, dropUserFeedEntryCollection, dropUser
```

peterzawistowicz/f26c4634dbbd5be9fcb0/raw/fa6b205413eb381cc55d7acac90b40c1b332bdf4/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/f26c4634dbbd5be9fcb0#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Frisby was briefly mentioned earlier. We will use this to define our test cases, as follows.

file name: **test/create*accounts*error_spec.js**

```
TU1_FN = "Test";
TU1_LN = "User1";
TU1_EMAIL = "testuser1@example.com";
TU1_PW = "testUser123";
TU_EMAIL_REGEX = 'testuser*';
SP_APP_NAME = 'Reader Test';
```

```

var frisby = require('frisby');
var tc = require('./config/test_config');
```

eterzawistowicz/fd461debff04c0d70bd8/raw/38b8b9f510d4c75257da51844874aad8112c9195/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/fd461debff04c0d70bd8#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

We will start with the **enroll route** in the following code. In this case we are deliberately missing the first name field, so we expect a status reply of 400 with a JSON error that we forgot to define the first name. Let's "toss that frisby":

```
frisby.create('POST missing firstName')
    .post(tc.url + '/user/enroll',
          { 'lastName' : TU1_LN,
            'email' : TU1_EMAIL,
            'password' : TU1_PW })
    .expectStatus(400)
    .expectHeader('Content-Type', 'application/json; charset=utf-8')
    .expectJSON({'error' : 'Undefined First Name'})
    .toss()
```

eterzawistowicz/4ea07cc0a27dcad2d262/raw/9eef8c0c55bae5417c086f01e5dbd54c08b2628b/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/4ea07cc0a27dcad2d262#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In the following example, we are testing a password that does not have any lower-case letters. This would actually result in an error being returned by Stormpath, and we would expect a status reply of 400.

```
frisby.create('POST password missing lowercase')
    .post(tc.url + '/user/enroll',
          { 'firstName' : TU1_FN,
            'lastName' : TU1_LN,
            'email' : TU1_EMAIL,
            'password' : 'TESTUSER123' })
    .expectStatus(400)
    .expectHeader('Content-Type', 'application/json; charset=utf-8')
    .expectJSONTypes({'error' : String})
    .toss()
```

eterzawistowicz/cdc7cc9f22cfab68cfd3/raw/5be6f18e4e732147bca68e200e0ca7ae13f52d23/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/cdc7cc9f22cfab68cfd3#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In the following example, we are testing an invalid email address. So, we can see that there is no @ sign and no domain name in the email address we are passing, and we would expect a status reply of 400.

```
frisby.create('POST invalid email address')
    .post(tc.url + '/user/enroll',
          { 'firstName' : TU1_FN,
            'lastName' : TU1_LN,
            'email' : "invalid.email",
            'password' : 'testUser' })
    .expectStatus(400)
    .expectHeader('Content-Type', 'application/json; charset=utf-8')
    .expectJSONTypes({'error' : String})
    .toss()
```

eterzawistowicz/39a9596ed190bf1c925a/raw/8c95463be56d669ac1a360eeec1ca63d900ff973/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/39a9596ed190bf1c925a#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Now, let’s look at some examples of test cases that should work. Let’s start by defining 3 users.

file name: **test/create*accounts*spec.js**

```
TEST_USERS = [{'fn' : 'Test', 'ln' : 'User1',
               'email' : 'testuser1@example.com', 'pwd' : 'testUser123'},
              {'fn' : 'Test', 'ln' : 'User2',
               'email' : 'testuser2@example.com', 'pwd' : 'testUser123'},
              {'fn' : 'Test', 'ln' : 'User3',
               'email' : 'testuser3@example.com', 'pwd' : 'testUser123'}]

SP_APP_NAME = 'Reader Test';

var frisby = require('frisby');
var tc = require('./config/test_config');
```

/peterzawistowicz/760cb9928f28e1e3cfab/raw/9913341c50b7cf8d6b7963b19ff0a71136cfde7e/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/760cb9928f28e1e3cfab#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In the following example, we are sending the array of the 3 users we defined above and are expecting a success status of 201. The JSON document returned would show the user object created, so we can verify that what was created matched our test data.

```
TEST_USERS.forEach(function createUser(user, index, array) {
    frisby.create('POST enroll user ' + user.email)
        .post(tc.url + '/user/enroll',
              { 'firstName' : user.fn,
                'lastName' : user.ln,
                'email' : user.email,
                'password' : user.pwd })
        .expectStatus(201)
        .expectHeader('Content-Type', 'application/json; charset=utf-8')
        .expectJSON({ 'firstName' : user.fn,
                      'lastName' : user.ln,
                      'email' : user.email })
        .toss()
});
```

eterzawistowicz/030de3cde5fcd1a2e130/raw/a20feab4898fdec2f3a6e2dec10ba0cde6c78395/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/030de3cde5fcd1a2e130#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we will test for a duplicate user. In the following example, we will try to create a user where the email address already exists.

```
frisby.create('POST enroll duplicate user ')
    .post(tc.url + '/user/enroll',
          { 'firstName' : TEST_USERS[0].fn,
            'lastName' : TEST_USERS[0].ln,
            'email' : TEST_USERS[0].email,
            'password' : TEST_USERS[0].pwd })
    .expectStatus(400)
    .expectHeader('Content-Type', 'application/json; charset=utf-8')
    .expectJSON({'error' : 'Account with that email already exists.  Please choos
    .toss()
```

n/peterzawistowicz/c95f207302adfdd69f4f/raw/d891d734fc0408671d647af13f3caed6a66f1e3b/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/c95f207302adfdd69f4f#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

One important issue is that we don’t know what API key will be returned by Stormpath a priori. So, we need to create a file dynamically that looks like the following. We can then use this file to define test cases that require us to authenticate a user.

file name: **/tmp/readerTestCreds.js**

```
TEST_USERS =
[{      "_id":"54ad6c3ae764de42070b27b1",
        "email":"testuser1@example.com",
        "firstName":"Test",
        "lastName":"User1",
        "sp_api_key_id":”<API KEY ID>",
        "sp_api_key_secret":”<API KEY SECRET>”
},
{       "_id":"54ad6c3be764de42070b27b2”,
        "email":"testuser2@example.com",
        "firstName":"Test",
        "lastName":"User2”,
        "sp_api_key_id":”<API KEY ID>",
        "sp_api_key_secret":”<API KEY SECRET>”
}];
module.exports = TEST_USERS;
```

peterzawistowicz/6e9e0b2da5c312ed546f/raw/e423e880aba69cb140f5867cdd6fcd1f7a3f0e85/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/6e9e0b2da5c312ed546f#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In order to create the temporary file above, we need to connect to MongoDB and retrieve user information. This is achieved by the following code.

file name: **tests/writeCreds.js**

```
TU_EMAIL_REGEX = new RegExp('^testuser*');
SP_APP_NAME = 'Reader Test';
TEST_CREDS_TMP_FILE = '/tmp/readerTestCreds.js';

var async = require('async');
var dbConfig = require('./config/db.js');
var mongodb = require('mongodb');
assert = require('assert');

var mongoClient = mongodb.MongoClient
var reader_test_db = null;
var users_array = null;
```

```

function connectDB(callback) {
        mongoClient.connect(dbConfig.testDBURL, function(err, db) {
                assert.equal(null, err);
                reader_test_db = db;
                callback(null);
        });
 }

 function lookupUserKeys(callback) {
        console.log("lookupUserKeys");
        user_coll = reader_test_db.collection('user');
        user_coll.find({email : TU_EMAIL_REGEX}).toArray(function(err, users) {
                users_array = users;
                callback(null);
        });
 }

function writeCreds(callback) {
        var fs = require('fs');
        fs.writeFileSync(TEST_CREDS_TMP_FILE, 'TEST_USERS = ');
        fs.appendFileSync(TEST_CREDS_TMP_FILE, JSON.stringify(users_array));
        fs.appendFileSync(TEST_CREDS_TMP_FILE, '; module.exports = TEST_USERS;');
        callback(0);
 }

 function closeDB(callback) {
        reader_test_db.close();
 }

 async.series([connectDB, lookupUserKeys, writeCreds, closeDB]);
```

terzawistowicz/8769632140f657e8e8b8/raw/083e69ac3665edb611e92853a60e509c588406ae/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/8769632140f657e8e8b8#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In the following code, we can see that the first line uses the temporary file that we created with the user information. We have also defined several feeds, such as Dilbert and the Eater Blog.

file name: **tests/feed_spec.js**

```

TEST_USERS = require('/tmp/readerTestCreds.js');

var frisby = require('frisby');
var tc = require('./config/test_config');
var async = require('async');
var dbConfig = require('./config/db.js');

var dilbertFeedURL = 'http://feeds.feedburner.com/DilbertDailyStrip';
var nycEaterFeedURL = 'http://feeds.feedburner.com/eater/nyc';
```

/peterzawistowicz/2a7631fda70b488858ff/raw/c8957bd9fd9ae189994291e4fc2139cc6100f9e8/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/2a7631fda70b488858ff#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Previously, we defined some users but none of them had subscribed to any feeds. In the following code we test feed subscription. Note that authentication is required now and this is achieved using .auth with the Stormpath API keys. Our first test is to check for an empty feed list.

```
function addEmptyFeedListTest(callback) {
        var user = TEST_USERS[0];
        frisby.create('GET empty feed list for user ' + user.email)
             .get(tc.url + '/feeds')
             .auth(user.sp_api_key_id, user.sp_api_key_secret)
             .expectStatus(200)
             .expectHeader('Content-Type', 'application/json; charset=utf-8')
             .expectJSON({feeds : []})
             .toss()
             callback(null);
}
```

peterzawistowicz/6cc4c105c66db12a4c3f/raw/ae9c1854a6abe8223bbec5bfef7f4afeba9d41d6/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/6cc4c105c66db12a4c3f#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In our next test case, we will subscribe our first test user to the Dilbert feed.

```
function subOneFeed(callback) {
        var user = TEST_USERS[0];
        frisby.create('PUT Add feed sub for user ' + user.email)
             .put(tc.url + '/feeds/subscribe', {'feedURL' : dilbertFeedURL})
             .auth(user.sp_api_key_id, user.sp_api_key_secret)
             .expectStatus(201)
             .expectHeader('Content-Type', 'application/json; charset=utf-8')
             .expectJSONLength('user.subs', 1)
             .toss()
```

```
                callback(null);
}
```

eterzawistowicz/a4c561acf5510585d5a6/raw/23906b43f9b01c4d94421a763e4b026afb70c607/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/a4c561acf5510585d5a6#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In our next test case, we will try to subscribe our first test user to a feed that they are already subscribed-to.

```
function subDuplicateFeed(callback) {
        var user = TEST_USERS[0];
        frisby.create('PUT Add duplicate feed sub for user ' + user.email)
             .put(tc.url + '/feeds/subscribe',
                  {'feedURL' : dilbertFeedURL})
             .auth(user.sp_api_key_id, user.sp_api_key_secret)
             .expectStatus(201)
             .expectHeader('Content-Type', 'application/json; charset=utf-8')
             .expectJSONLength('user.subs', 1)
             .toss()
        callback(null);
}
```

eterzawistowicz/7aa3c443ff5cb77f663b/raw/bf867110c2741861501565db7d13e90e43b8b97c/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/7aa3c443ff5cb77f663b#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we will subscribe our test user to a new feed. The result returned should confirm that the user is subscribed now to 2 feeds.

```
function subSecondFeed(callback) {
        var user = TEST_USERS[0];
        frisby.create('PUT Add second feed sub for user ' + user.email)
             .put(tc.url + '/feeds/subscribe',
                  {'feedURL' : nycEaterFeedURL})
             .auth(user.sp_api_key_id, user.sp_api_key_secret)
             .expectStatus(201)
             .expectHeader('Content-Type', 'application/json; charset=utf-8')
             .expectJSONLength('user.subs', 2)
             .toss()
        callback(null);
 }
```

/peterzawistowicz/742bce43b214f8c4265e/raw/a9b3127c5fd560c0e4b13f3f874f9ff93f90c03c/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/742bce43b214f8c4265e#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Next, we will use our second test user to subscribe to a feed.

```
function subOneFeedSecondUser(callback) {
        var user = TEST_USERS[1];
        frisby.create('PUT Add one feed sub for second user ' + user.email)
             .put(tc.url + '/feeds/subscribe',
                  {'feedURL' : nycEaterFeedURL})
             .auth(user.sp_api_key_id, user.sp_api_key_secret)
             .expectStatus(201)
             .expectHeader('Content-Type', 'application/json; charset=utf-8')
             .expectJSONLength('user.subs', 1)
             .toss()
        callback(null);
}

async.series([addEmptyFeedListTest, subOneFeed, subDuplicateFeed, subSecondFeed,
```

eterzawistowicz/87ed5f0c83079c90dfe8/raw/22bcbb42763837c27c420bc4cfc257f5406d8c2c/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/87ed5f0c83079c90dfe8#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

The REST API

Before we begin writing our REST API code, we need to define some utility libraries. First, we need to define how our application will connect to the database. Putting this information into a file gives us the flexibility to add different database URLs for development or production systems.

file name: **config/db.js**

```
module.exports = {
     url : 'mongodb://localhost/reader_test'
 }
```

eterzawistowicz/88b8581bb4b94dc053c3/raw/904657cdaa659b29756690f0fb6b3dfc48d61064/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/88b8581bb4b94dc053c3#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

If we wanted to turn on database authentication we could put that information in a file, as shown below. This file should not be checked into source code control for obvious reasons.

file name: **config/security.js**

```
module.exports = {
     stormpath_secret_key : ‘YOUR STORMPATH APPLICATION KEY’;
}
```

eterzawistowicz/36daed56192f2dfce29b/raw/61957fe5726e300910d9dc264ec9d4fea365478c/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/36daed56192f2dfce29b#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

We can keep Stormpath API and Secret keys in a properties file, as follows, and need to carefully manage this file as well.

file name: **config/stormpath_apikey.properties**

```
apiKey.id = YOUR STORMPATH API KEY ID
apiKey.secret = YOUR STORMPATH API KEY SECRET
```

eterzawistowicz/cfd6db7a42f650793492/raw/e8e73c7e3adce914b9f6b5efa13ea9b51835b8e6/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/cfd6db7a42f650793492#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Express.js overview

In Express.js, we create an “application” (app). This application listens on a particular port for HTTP requests to come in. When requests come in, they pass through a middleware chain. Each link in the middleware chain is given a req (request) object and a res (results) object to store the results. Each link can choose to do work, or pass it to the next link. We add new middleware via `app.use()`. The main middleware is called our “router”, which looks at the URL and routes each different URL/verb combination to a specific handler function.

Creating our application

Now we can finally see our application code, which is quite small since we can embed handlers for various routes into separate files.

file name: **server.js**

```
var express = require('express');
var bodyParser = require('body-parser');
var mongoose = require('mongoose');
var stormpath = require('express-stormpath');
var routes = require("./app/routes");
var db   = require('./config/db');
var security = require('./config/security');

var app = express();
var morgan = require('morgan’);
app.use(morgan);
app.use(stormpath.init(app, {
     apiKeyFile: './config/stormpath_apikey.properties',
     application: ‘YOUR SP APPLICATION URL',
     secretKey: security.stormpath_secret_key
 }));
 var port = 8000;
 mongoose.connect(db.url);

app.use(bodyParser.urlencoded({ extended: true }));

routes.addAPIRouter(app, mongoose, stormpath);
```

eterzawistowicz/be7091aae3e32aed0183/raw/2944608ee039ea4e1f6d69c692abbbba41722af7/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/be7091aae3e32aed0183#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

We define our own middleware at the end of the chain to handle bad URLs.

```
app.use(function(req, res, next){
   res.status(404);
   res.json({ error: 'Invalid URL' });
});
```

erzawistowicz/1c229dde0d61ebd571bc/raw/c9892e03cd23b45d404758316449cab7605bba61/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/1c229dde0d61ebd571bc#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Now our server application is listening on port 8000.

```
app.listen(port);
```

peterzawistowicz/643827e9dadae1ad9789/raw/b36617b1f6611f90bc63f5375f6d3c662a9178fa/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/643827e9dadae1ad9789#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Let’s print a message on the console to the user.

```
console.log('Magic happens on port ' + port);

exports = module.exports = app;
```

eterzawistowicz/b58561ce2787c46b0304/raw/3839006d74ced4e2d90eac06cefd2834e08fd32c/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/b58561ce2787c46b0304#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Defining our Mongoose data models

We use Mongoose to map objects on the Node.js side to documents inside MongoDB. Recall that earlier, we defined 4 collections:

1. Feed collection.
2. Feed entry collection.
3. User collection.
4. User feed-entry-mapping collection.

So we will now define schemas for these 4 collections. Let’s begin with the user schema. Notice that we can also format the data, such as converting strings to lowercase, and remove leading or trailing whitespace using trim.

file name: **app/routes.js**

```
var userSchema = new mongoose.Schema({
         active: Boolean,
         email: { type: String, trim: true, lowercase: true },
         firstName: { type: String, trim: true },
         lastName: { type: String, trim: true },
         sp_api_key_id: { type: String, trim: true },
         sp_api_key_secret: { type: String, trim: true },
         subs: { type: [mongoose.Schema.Types.ObjectId], default: [] },
         created: { type: Date, default: Date.now },
         lastLogin: { type: Date, default: Date.now },
     },
     { collection: 'user' }
);
```

/peterzawistowicz/7be496a1f23370cffdd3/raw/6c3d8a6afb5ff313b67e3983ad18828410e945f1/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/7be496a1f23370cffdd3#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

In the following code, we can also tell Mongoose what indexes need to exist. Mongoose will also ensure that these indexes are created if they do not already exist in our MongoDB database. The unique constraint ensures that duplicates are not allowed. The “email : 1” maintains email addresses in ascending order. If we used “email : -1” it would be in descending order.

```
userSchema.index({email : 1}, {unique:true});
userSchema.index({sp_api_key_id : 1}, {unique:true});
```

eterzawistowicz/841df079b512cc84785a/raw/cf35d03c76d798ba3b7e2a22a0af35e2b2f48d21/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/841df079b512cc84785a#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

We repeat the process for the other 3 collections.

```
var UserModel = mongoose.model( 'User', userSchema );

var feedSchema = new mongoose.Schema({
         feedURL: { type: String, trim:true },
         link: { type: String, trim:true },
         description: { type: String, trim:true },
         state: { type: String, trim:true, lowercase:true, default: 'new' },
         createdDate: { type: Date, default: Date.now },
         modifiedDate: { type: Date, default: Date.now },
     },
     { collection: 'feed' }
);

feedSchema.index({feedURL : 1}, {unique:true});
feedSchema.index({link : 1}, {unique:true, sparse:true});

var FeedModel = mongoose.model( 'Feed', feedSchema );

var feedEntrySchema = new mongoose.Schema({
         description: { type: String, trim:true },
         title: { type: String, trim:true },
         summary: { type: String, trim:true },
         entryID: { type: String, trim:true },
         publishedDate: { type: Date },
         link: { type: String, trim:true  },
         feedID: { type: mongoose.Schema.Types.ObjectId },
         state: { type: String, trim:true, lowercase:true, default: 'new' },
         created: { type: Date, default: Date.now },
     },
     { collection: 'feedEntry' }
);

feedEntrySchema.index({entryID : 1});
feedEntrySchema.index({feedID : 1});

var FeedEntryModel = mongoose.model( 'FeedEntry', feedEntrySchema );

var userFeedEntrySchema = new mongoose.Schema({
         userID: { type: mongoose.Schema.Types.ObjectId },
         feedEntryID: { type: mongoose.Schema.Types.ObjectId },
         feedID: { type: mongoose.Schema.Types.ObjectId },
         read : { type: Boolean, default: false },
     },
     { collection: 'userFeedEntry' }
 );
```

n/peterzawistowicz/6fed36bed62cf990a5f7/raw/68effc42fe304dd70faedf7621d2ce68ad86f10b/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/6fed36bed62cf990a5f7#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

The following is an example of a compound index on 4 fields. Each index is maintained in ascending order.

```
userFeedEntrySchema.index({userID : 1, feedID : 1, feedEntryID : 1, read : 1});

var UserFeedEntryModel = mongoose.model('UserFeedEntry', userFeedEntrySchema );
```

peterzawistowicz/54f82d1d2fe92fc1621d/raw/cc07a48d146fb12782574faac4b8d3667e929663/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/54f82d1d2fe92fc1621d#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Every route that comes in for `GET`, `POST`, `PUT` and `DELETE` needs to have the correct content type, which is **application/json**. Then the next link in the chain is called.

```
exports.addAPIRouter = function(app, mongoose, stormpath) {

        app.get('/*', function(req, res, next) {
                res.contentType('application/json');
                next();
        });
        app.post('/*', function(req, res, next) {
                res.contentType('application/json');
                next();
        });
        app.put('/*', function(req, res, next) {
                res.contentType('application/json');
                next();
        });
        app.delete('/*', function(req, res, next) {
                res.contentType('application/json');
                next();
        });
```

peterzawistowicz/2d26db56a0720e907bfc/raw/90217866b2f2740fbad253d73c61a2ff5929d923/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/2d26db56a0720e907bfc#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Now we need to define handlers for each combination of URL/verb. The link to the complete code is available in the resources section and we just show a few examples below. Note the ease with which we can use Stormpath. Furthermore, notice that we have defined **/api/v1.0**, so the client would actually call **/api/v1.0/user/enroll**, for example. In the future, if we changed the API, say to 2.0, we could use **/api/v2.0**. This would have its own router and code, so clients using the v1.0 API would still continue to work.

```
var router = express.Router();

        router.post('/user/enroll', function(req, res) {
                logger.debug('Router for /user/enroll');
                …
        }
        router.get('/feeds', stormpath.apiAuthenticationRequired, function(req, r
                logger.debug('Router for /feeds');
                …
        }
        router.put('/feeds/subscribe',
                        stormpath.apiAuthenticationRequired, function(req, res)
                logger.debug('Router for /feeds');
                …
        }
        app.use('/api/v1.0', router);
}
```

terzawistowicz/a2075d558582d3fd5d2c/raw/862cce11b85879eb975a066654a9729428292e9e/gistfile1.txt)
gistfile1.txt (https://gist.github.com/peterzawistowicz/a2075d558582d3fd5d2c#file-gistfile1-txt) hosted with ❤ by **GitHub (https://github.com)**

Starting the server and running tests

Finally, here is a summary of the steps we need to follow to start the server and run the tests.

- Ensure that the MongoDB instance is running
    - mongod
- Install the Node libraries
    - npm install
- Start the REST API server
    - node server.js
- Run test cases
    - node setup_tests.js
    - jasmine-node create_accounts_error_spec.js
    - jasmine-node create_accounts_spec.js
    - node write_creds.js
    - jasmine-node feed_spec.js

MongoDB University (https://university.mongodb.com/) provides excellent free training. There is a course specifically aimed at Node.js developers (https://university.mongodb.com/courses/M101JS/about?_ga=1.197810283.515780760.1428941165) and the link can be found in the resources section below. The resources section also contains links to good MongoDB data modeling resources.

Resources

HTTP status code definitions (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)

Chad Tindel's Github Repository (https://github.com/ctindel/reader)

M101JS: MongoDB for Node.js Developers (https://university.mongodb.com/courses/M101JS/about)

Data Models (http://docs.mongodb.org/manual/data-modeling/)

Data Modeling Considerations for MongoDB Applications (http://docs.mongodb.org/v2.2/core/data-modeling/)

Learn more MongoDB. Register for our free online courses:

Learn MongoDB for free **(https://university.mongodb.com/?jmp=blog)**

<< Read Part 1 (https://www.mongodb.com/blog/post/building-your-first-application-mongodb-creating-rest-api-using-mean-stack-part-1)

About the Author - Norberto

Norberto Leite is Technical Evangelist at MongoDB. Norberto has been working for the last 5 years on large scalable and distributable application environments, both as advisor and engineer. Prior to MongoDB Norberto served as a Big Data Engineer at Telefonica.

Search

Featured

3.0 (/blog/post/whats-new-mongodb-30-part-1-95-reduction-operationaloverhead-and-security-enhancements)

Events
(//www.mongodb.com/blog/post/spend-an-evening-with-mongodb)

Spark
(//www.mongodb.com/blog/post/leaf-in-the-wild-stratio-integrates-apache-spark-and-mongodb-to-unlock-new-customer-insights-for-one-of-worlds-largest-banks)

Community
(//www.mongodb.com/blog/post/now-accepting-nominations-for-the-2015-william-zola-award-for-community-excellence)

Security
(//www.mongodb.com/blog/post/july-mongodb-security-best-practices)

Ops Best Practices
(//www.mongodb.com/blog/post/your-ultimate-guide-to-rolling-upgrades)

News
(//www.mongodb.com/blog/post/dbta-presents-mongodb-readers-choice-award-for-second-year-in-row)

Receive updates from MongoDB

Business email:

First name:

Last name:

Business phone:	
Company:	
Job function:	-- Please Select --
Country:	-- Please Select --
State:	Not Applicable

Subscribe

4 Comments MongoDB.com Blog 1 Login

Recommend 4 Share Sort by Newest

Join the discussion…

Javier • 16 days ago

Hi there!

As AsGiants have stated, I'm having troubles finding the resources section for the GitHub repository. I'd like to have a look at it.

Reply • Share ›

AsGiants Astros • 2 months ago

Also, I don't understand where angular/json responses plays into the application. I assume that angular interprets json and generates html pages? I tried to look at the js files but couldn't find them, hence my previous comment

Reply • Share ›

AsGiants Astros • 2 months ago

Hi, thanks for the article. Is there a link to the github repo? I'm having trouble finding the resources section for the blog. There seems to be no links for code or code repositories from either the 'Learn' section of the mongodb website, or from the blog itself (just listings of posts/search for posts).

Reply • Share ›

Sergio Henrique • 6 months ago

Great article!I needed to do something today and were able to help me! Keep up post's talking about geolocation is a very nice subject and has few articles on the web.

Reply • Share ›

ALSO ON MONGODB.COM BLOG WHAT'S THIS?

Announcing MongoDB's Giant of the Month, Attila Toszer of Amadeus

11 comments • 5 months ago

Sig Narvaez — Congrats!!

How We Brought the MongoDB University App to Life

3 comments • 5 months ago

Andrew Shapton — Looking forward to the Android app and a bunch of new announcements !

Analyzing Data in Microsoft Excel with the MongoDB Connector for BI

2 comments • 2 months ago

Andrew Morgan — Murat – any BI tool that uses SQL to query a data store should work with the new BI connector. As an …

What's New in MongoDB 3.2, Part 1: Extending Use Cases with New …

3 comments • 3 months ago

Douglas Duncan — Thanks for the great write up of the new 3.2 features Mat! I'm looking forward to getting in and testing …

Subscribe Add Disqus to your site Add Disqus Add Privacy

About MongoDB

About MongoDB, Inc (/company)

Careers (/careers)

Contact Us (/contact)

Legal Notices (/legal/legal-notices)

Security Information (/security)

Learn More

NoSQL Databases Explained (/nosql-explained)

MongoDB Architecture Guide (/lp/white-paper/architecture-guide)

MongoDB Enterprise Advanced (/products/mongodb-enterprise-advanced)

MongoDB Cloud Manager (/cloud)

Engineering @ MongoDB (https://engineering.mongodb.com)