Notes on Express.js

Ravi Shankar 20124025

Table of Contents

1. Creating first express app

Run the following command to install express-generator on your system. The following command should be preceded by `sudo` on ubuntu and mac.

```
npm install express-generator -g
```

Now, reach a folder where you would like to make your first node app. And inside that folder run the following command in the order specified.

```
express firstApp --ejs
cd firstApp
npm install
```

If you are facing proxy error while doing npm install, then do the following.

```
gedit ~/.npmrc
```

A file will open in gedit, now write the following content in that file and save

```
proxy=http://edcguest:edcguest@172.31.103.29:3128/
https-proxy=http://edcguest:edcguest@172.31.103.29:3128/
```

This will work for Linux, Ubuntu. Windows user can access my blog[1] for detailed instruction.

Now, run the following command from terminal `npm start` and open a browser and visit `localhost:3000`

If you were welcomed by express, pat yourself on back and thank me.

Folder Structure of express has already been discussed, so now directly come to the routing part.

2. Routing in express app

Look at the following code.

routes/index.js

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});
```

This code means that when we visit `/` i.e. `localhost:3000` in our current app, it will call this function to handle the request.

```
function(req, res, next) {
  res.render('index', { title: 'Express' });
}
```

[1] https://ilearncomputer.wordpress.com/2015/05/10/using-npm-behind-proxy/

This function takes three arguments, `req`, `res` and next.

`req` contains the details about the request.

`res` contains the details about the server response.

`next` represents the next middleware. middleware will be explained later.

This line,

```
res.render('index', { title: 'Express' });
```

means that, index.ejs(notice the first argument of the render method) file will be compiled and sent to the client. Second argument of the render function, is a set of variables that the render sends to the engine that compiles `index.ejs`. As you can see that, this is a way to send some data to the compiler so that the compiler can modify the page or insert/delete some information in the page depending upon the data it receives from the render function.

```
Add a router such that, when someone opens localhost:3000/me, it displays
 a page with your name.
```

To achieve the above task, add following code to the `routes/index.js` file

routes for `/me`

```
router.get('/me', function(req, res, next) {
  res.render('me', { title: 'about me', name: 'Ravi Shankar' });
});
```

And add a file `me.ejs` to the views folder with the following content

views/me.ejs

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title> <%= title %> </title>
  </head>
  <body>
    My name is : <%= name %>
  </body>
```

```
</html>
```

You can clearly see that the variables that we pass from routes cab be accessed in the page using `<%= variable %>` syntax.

Now lets take an example in which if a user tries to access `/year/number`, where number denotes his/her year at college(1/2/3/4). And then show a custom message depending upon the year.

Add following routes to the router.

Add these lines to routes/index.js

```
router.get('/year/:id', function(req, res, next) {
  var year = req.params.id;
  if(year == 1){
    res.send("Too much energy");
  } else if(year ==2){
    res.send("Learning to be lazy");
  } else if (year == 3){
    res.send("Intern---intern---placement--placement");
  } else if(year==4){
    res.send("laziest people on the campus");
  } else {
    res.send("tumhara kuchh nahi ho sakta :P");
  }
});
```

```
Assignment: Look for an javascript function that will parse req.params.id
 into a number. By, default req.params.id is not a number. We were able to
 compare because javascript has soft equility operator.
```

3. mongoose.js

I can safely assume that you guys have installed node.js, mongoDB, and know how to start mongoDB and you are able to install packages through `npm install` command.

3.1. Introduction

In this section we are going to learn how to use mongoose for managing database in node.js application

What is mongoose?

Mongoose is a middleware that we use for connecting to the database in node.js application

Why mongoose?

mongoDB is a schemaless database, but, we need at least a little bit of structure in our document. Mongoose achieves the same in a very simple and easy way.

what are we going to learn?

We will learn how to perform CRUD operation on Database.

3.2. Connecting to database

First of all you need to switch on mongodb if it is already not on.

Switching on mongodb

- Download the mongodb from the website
- unzip the file
- Enter into the mongodb folder
- Again enter into the mongodb folder
- Enter into the bin folder
- Now make folder in this place named database
- Now give the command `mongod --dbpath=database`

> On windows, it may ask for permission before actually starting.

Connecting the app to database

Before we can use mongoose, it must be installed and included in the app.

Install mongoose by the following command

```
npm install mongoose --save
```

To include mongoose, using the following code in app.js

Add these lines in app.js

```
var mongoose = require('mongoose');
```

Now, since we have included mongoose in app.js, we can use use mongoose to connect with the database.

To connect with the database, add the following lines in app.js.

Add the following lines in app.js

```
mongoose.connect('mongodb://localhost/firstApp');
```

This will use firstApp database, if it exists, otherwise it will create one.

If you don't connect the app to the database, mongoose may seem to work fine, but it won't be able to save the objects to the database.

3.3. Making an schema

What is schema?

Schema can be thought of as structure of objects that we are going to represent in database.

Example 1. Schema

If we are trying to represent a product in database. Then we may represent the product as an object with two properties namely price and name.

Create schema for a product

Inside the routes folder, make a folder named `models`, in this folder, we will define schema for different entities such as users and products. Inside the model folder, make a file `product.js`. In `product.js` file, we will write schema for products.

Write the following code in product.js

```
var mongoose = require('mongoose');
```

```
var ObjectId = mongoose.Schema.Types.ObjectId;

var schema = mongoose.Schema({
  name      : {type: String, default: 'Stylish T-shirt'},
  price     : {type: Number, default : 499}
});

module.exports = mongoose.model('Product', schema);
```

3.4. Using the schema created above

In the `index.js` file, add the following code

Add the following code in index.js file

```
var Product = require('./models/product.js');
```

This will import product schems in `index.js` file, in `Product` variable. Now, we can use `Product` to create, read, update and delete objects of product type.

3.5. Creating an object(Product)

Creating a product is as simple as

```
var p = new Product();
```

Add these lines in the routes, where you get the request for creating a new product.

Although product has been created, but we have not saved any information in it. Our schem has two fields namely name and price, so lets give these attributes some value.

```
p.name = "My stylish shirt";
p.price = 699;
```

Above code will set the values of given attribute. To save the changes in database, we will use the following code.

```
p.save();
```

The code for handling request for adding new product(in `index.js`) should now look like

```
router.get('/addnewproduct', function(req, res, next) {
  var p = new Product();
  p.name = "My stylish shirt";
  p.price = 699;
  p.save();
  res.redirect('/');
});
```

```
Assignment: The last route doesn't guarantee that the product were
 actually added or not. As an assignment, look for save() api on mongoose
 website and implement the render function inside callback of p.save();
```

3.6. Reading an object

Reading an object is very simple. Mongoose provides two api `find()` and `findOne()` to search for products in its database.

Look at the following function.

Add the following code to `index.js` to get all the entries of Product type

```
router.get('/allProduct', function(req, res, next) {
  Product.find(function(err,doc){
    if(err){
      //TODO
    };
    res.send(doc);
  });
});
```

When we go to */allProduct*, we get an array of product. Now copy the id of any one product and visit the following address `/product/id`, replace id with the id you copied.

Wait, let us add the route for that first.

```
router.get('/product/:id', function(req, res, next) {
  Product.findOne(function(err,doc){
    if(err){
      //TODO
    });
    res.send(doc.name);
```

```
});
```

Until now, we have just been asking the server to add the product for us. And it has been adding product for us with hard coded names and hard coded prices.

Its time to correct this.

I think, its a good time to introduce to you guys the body parser. Open `app.js file`. You will see some code like this.

```
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
```

They actually parse the body of the request, and save the data in `req.body`. For example, if you submit the form with two fields, name and price, using a post protocol. Bodyparser will read the post request, and save the entries in req.body. `name` will be saved in `req.body.name`. And `price` will be saved in `req.body.price`.

Lets add a form in `views` to add product.

views/addcustomproduct.ejs

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title> <%= title %> </title>
  </head>
  <body>
    <form action="/addcustomproduct" method="post">
        <div class="form-group">
            <label>Product Name</label>
            <input type="text" class="form-control" name="productname">
        </div>
        <div class="form-group">
            <label>Price</label>
            <input type="number" class="form-control" name="price">
        </div>

        <button type="submit" class="btn btn-warning btn-lg">Add The
 product</button>
    </form>
  </body>
</html>
```

Lets also add a route for viewing this form and a route for adding custom product.

```
router.get('/addcustomproduct', function(req, res, next) {
  res.render('addcustomproduct', { title: 'add new product' });
});

router.post('/addcustomproduct', function(req, res, next) {
  var p = new Product();
  if(req.body.productname){
    p.name=req.body.productname;
  };
  if(req.body.price){
    p.price=req.body.price;
  };
  p.save(function(err){
    if(!err){
      console.log("Data has been saved");
    } else {
      console.log("Error occurred");
      res.redirect('/');
    }
  });
    res.redirect('/');
});
```

When we post the data, it is sent with req. The body-parsers will read the data and then save them in req object.

3.7. Updating an object

Updating an object is as simple as finding an object, modifying it and saving it. The following code will suffice.

```
Product.findOne({_id: req.param("productId")}, function(err, doc){
  doc.name = "myProduct";
  doc.save();
});
```

Add the following code to the index.js to see update function in action

```
router.get('/update/:productId', function(req, res, next) {
    var productId = req.param("productId");
    Product.findOne({_id: req.param("productId")}, function(err, doc){
```

```
      doc.name = "Modified product";
      doc.save();
    });
    res.redirect('/');
});
```

Update a few product. Now, run `localhost:3000/allproduct` to see the changes.

3.8. Deleting an object

It is very similar to find function. Watch out for the following code

```
Product.remove({_id: req.param("productId")}, function(err){
  console.log("removed");
});
```

Add the following code to the index.js to see delete function in action

```
router.get('/delete/:productId', function(req, res, next) {
    var productId = req.param("productId");
    Product.remove({_id: req.param("productId")}, function(err){
      console.log("removed");
    });
    res.redirect("/");
});
```

Try deleting a few product. Now, run `localhost:3000/allproduct` to see the changes.

This is the end of mongoose section.

4. Passport

4.1. Introduction

Passport is a library that manage authentication in a node.js web-app. In the name of authentication it manages login, sign-up, logout activities. These features can be written from grond-up, yet people use passport because, passport is time-tested and makes development process faster and seamless. Using passport in a node.js web-app is a no-brainer.

4.2. How to use passport?

Installing

Go to the app folder and run the following command.

```
npm install passport --save
```

We, will also be using, `bcrypt-nodejs` and `passport-local`. So, run the following command:

```
npm install bcrypt-nodejs --save
```

```
npm install passport-local --save
```

```
npm install connect-flash --save
```

```
npm install express-session --save
```

Well, if you don't have internet connection, you can copy the `passport`, `passport-local`, `bcrypt-nodejs` folder from somewhere into the node_modules folder in the app directory. And update the package.json file.

Creating a schema for user

Everything that goes to database has an associated schema. Details of user will be saved to the database so, we need to create a schema for users. Moreover, passport will need a user schema to work.

How to create a schema for user?

In the last chapter, we created a schema for product, schema for users can be also created in similar manner.

Creating user schema

- go to routes folder
- go to models folder

- make a file, `user.js`

write following contents in `user.js`

routes/models/user.js

```
var mongoose = require('mongoose');
var ObjectId = mongoose.Schema.Types.ObjectId;

var bcrypt   = require('bcrypt-nodejs');

// define the schema for our user model
var userSchema = mongoose.Schema({

    local            : {
        email        : String,
        password     : String
    }

});

// methods ======================
// generating a hash
userSchema.methods.generateHash = function(password) {
    return bcrypt.hashSync(password, bcrypt.genSaltSync(8), null);
};

// checking if password is valid
userSchema.methods.validPassword = function(password) {
    return bcrypt.compareSync(password, this.local.password);
};

// create the model for users and expose it to our app
module.exports = { model: mongoose.model('UserInfo', userSchema), schema:
 userSchema }
```

> Schema is not a javascript object, although it looks similar.

A hash of password will be created and saved, so even if someone has access to the database, she/she will not be able to see your passport.

While signing in, the password that you use to log-in, will be converted into a hash first, and this hash value will be compared to the saved value.

Writing the passport strategy

Inside the `routes` folder make a file `passport-strategy.js` and write the following content in that file

routes/passport-strategy.js

```
var LocalStrategy   = require('passport-local').Strategy;
var User            = require('./models/user').model;

// expose this function to our app using module.exports
module.exports = function(passport) {

    //
 ============================================================================
    // passport session setup
 ================================================
    //
 ============================================================================
    // required for persistent login sessions
    // passport needs ability to serialize and unserialize users out of
 session

    // used to serialize the user for the session
    passport.serializeUser(function(user, done) {
        done(null, user.id);
    });

    // used to deserialize the user
    passport.deserializeUser(function(id, done) {
        User.findById(id, function(err, user) {
            done(err, user);
        });
    });

    //
 ============================================================================
    // LOCAL SIGNUP
 ================================================================
    //
 ============================================================================
    // we are using named strategies since we have one for login and one
 for signup
    // by default, if there was no name, it would just be called 'local'
```

```
    passport.use('local-signup', new LocalStrategy({
        // by default, local strategy uses username and password, we will
override with email
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire
request to the callback
    },
    function(req, email, password, done) {

        // asynchronous
        // User.findOne wont fire unless data is sent back
        process.nextTick(function() {

        // find a user whose email is the same as the forms email
        // we are checking to see if the user trying to login already
exists
        User.findOne({ 'local.email' :  email }, function(err, user) {
            // if there are any errors, return the error
            if (err)
                return done(err);

            // check to see if theres already a user with that email
            if (user) {
                return done(null, false, req.flash('signupMessage', 'That
email is already taken.'));
            } else {

                // if there is no user with that email
                // create the user
                var newUser            = new User();

                // set the user's local credentials
                newUser.local.email    = email;
                newUser.local.password = newUser.generateHash(password);

                // save the user
                newUser.save(function(err) {
                    if (err)
                        throw err;
                    return done(null, newUser);
                });
            }

        });
```

```
        });

    }));

    //
=========================================================================
    // LOCAL LOGIN
=============================================================
    //
=========================================================================
    // we are using named strategies since we have one for login and one
for signup
    // by default, if there was no name, it would just be called 'local'

    passport.use('local-login', new LocalStrategy({
        // by default, local strategy uses username and password, we will
override with email
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire
request to the callback
    },
    function(req, email, password, done) { // callback with email and
password from our form

        // find a user whose email is the same as the forms email
        // we are checking to see if the user trying to login already
exists
        User.findOne({ 'local.email' :  email }, function(err, user) {
            // if there are any errors, return the error before anything
else
            if (err)
                return done(err);

            // if no user is found, return the message
            if (!user)
                return done(null, false, req.flash('loginMessage', 'No
user found.')); // req.flash is the way to set flashdata using connect-
flash

            // if the user is found but the password is wrong
            if (!user.validPassword(password))
```

```
                return done(null, false, req.flash('loginMessage', 'Oops!
 Wrong password.')); // create the loginMessage and save it to session as
 flashdata

            // all is well, return successful user
            return done(null, user);
        });

    }));

};
```

The code that we wrote in passport.js is actually suggested to be written by passport.js support website. Visit passport.js support site for more detail.

Also note that, we are exporting a function. This is a very common practice in javaScript. The function that we are exporting is taking passport as an argument. So, while requiring `passport-strategy`, it will need to be called with passport as an argument

Example:

```
require('passport-strategy')(passport);
```

Serialize and de-serialize exists there to help us maintain persistent login sessions. Which means, if I have logged in to a site, and I refresh the site, I should still remain logged in. Try experimenting with deleting the serialize and de-serialize functions.

`nextTick()` function is a function that holds the execution of a block of statements till next cycle of event loop. Basically, it asks javascript to execute all other tasks currently present in event loop and then come back to it and execute.

Wouldn't it be great if all the people in queue at Railway Ticket counter and Banks were enclosed in a nextTick() function :P

Now include passport in app.js and then call passport-strategy We will also be including connect-flash and express-session.

Add these lines to the app.js

```
var session = require('express-session');
var passport = require('passport');
```

```
var flash    = require('connect-flash');

//requiring passport strategy
require('./routes/passport-strategy.js')(passport);
```

Notice that we are including the three node_modules.

Now, below `app.use(express.static(path.join(__dirname, 'public')));` line add the following code.

Add them to app.js

```
//for passport
app.use(session({
      secret: 'this-is-my-secret-i-wont-tell-u',
      resave: true,
      saveUninitialized: true
}));
app.use(passport.initialize());
app.use(passport.session());
app.use(flash());
```

`app.use` is a way to tell the express to use middlewares on our app. Above code asks the express to use four middlewares. They will be explained in the lecture.

Before we proceed further, its time to add login page and sign up page in views and add routes for them.

routes for login and sign up page

```
router.get('/login', function(req, res, next) {
  res.render('login', { title: 'Login to firstApp'});
});

router.get('/signup', function(req, res, next) {
  res.render('signup', { title: 'Sign Up for firstApp'});
});
```

Lets, now add the views.

views/login.ejs

```
<!doctype html>
<head><title><%= title %></title></head>
```

```
  <body>
    <!-- LOGIN FORM -->
    <form action="/login" method="post">
        <div>
            <label>Email</label>
            <input type="text" name="email">
        </div>
        <div>
            <label>Password</label>
            <input type="password" name="password">
        </div>

        <button type="submit">Login</button>
    </form>

    <hr>

    <p>Need an account? <a href="/signup">Signup</a></p>
    <p>Or go <a href="/">home</a>.</p>
  <body>
</html>
```

views/signup.ejs

```
<!doctype html>
<head><title><%= title %></title></head>
  <body>
    <!-- Sign up FORM -->
    <form action="/signup" method="post">
        <div>
            <label>Email</label>
            <input type="text" name="email">
        </div>
        <div>
            <label>Password</label>
            <input type="password" name="password">
        </div>

        <button type="submit">Signup</button>
    </form>

    <hr>

    <p>Already have an account? <a href="/login">Login</a></p>
    <p>Or go <a href="/">home</a>.</p>
  <body>
```

```
</html>
```

Now that we have added routes for getting signup/login page and also added login/signup page to the views. You can restart the app and visit `localhost:3000/login` and `localhost:3000/signup` to see the pages. Mind you that, we have not written back-end code for login/signup. Lets write them up.

To login/signup, we will be accessing passport module in index.js, so add these lines to include passport.

Add this line in /routes/index.js

```
var passport = require('passport');
```

4.3. Sign Up

Add the following code in index.js to enable sign-up.

```
router.post('/signup', passport.authenticate('local-signup', {
  successRedirect : '/profile', // redirect to the secure profile section
  failureRedirect : '/signup', // redirect back to the signup page if
there is an error
  failureFlash : true // allow flash messages
}));
```

4.4. Log In

Add the following code in index.js to enable log-in.

```
router.post('/login', passport.authenticate('local-login', {
  successRedirect : '/profile', // redirect to the secure profile section
  failureRedirect : '/login', // redirect back to the signup page if
there is an error
  failureFlash : true // allow flash messages
}));
```

4.5. Log Out

Add the following code in index.js to enable log-out.

```
router.get('/logout', function(req, res, next) {
```

```
    req.logout();
    res.redirect('/');
  });
```

Take note in the routes that on successful login/signup, we are redirecting the page to `/profile`. So, lets create a route and view for the same.

views/profile.ejs

```
<!doctype html>
<head><title><%= title %></title></head>
  <body>
    My email id is : <%= userEmail %>
  <body>
</html>
```

Add this route to routes/index.js

```
router.get('/profile', function(req, res, next) {
  if(req.user){
    res.render('profile', { title: 'Profile Page', userEmail:
 req.user.local.email });
  } else {
    res.render('profile', {title: "You are not logged in", userEmail: "Not
 Logged In"});
  }
});
```

Note the way, we are passing user email from routes to the template engine. If the user is logged in, the profile page will be customized according to user otherwise we will show a simple page.

For any query, call me: 7571872733 or drop an email: ravishankarkumar@live.com[2]

If you want this completed app, then visit clone or fork this repo[3] and you are good to go :)

Visit my home page here[4] and follow this link to visit NodeJS MNNIT community group[5]

[2] mailto:ravishankarkumar@live.com
[3] https://github.com/mnnit-workspace/mongoose-passport-teaching-app
[4] http://www.ravishankarkumar.com
[5] https://www.facebook.com/groups/nodejsmnnit