Web Technologies: Server Report

Ben Crabbe

University of Bristol, UK

June 5, 2015

1 Introduction

My server implements the following features

- https self signed authentication
- user accounts with stored session info
- mongodb for storing users and user's posts
- encrypted password storage
- dynamic front page with searching/filtering system using the tags search bar and links
- comments system
- all pages delivered through dynamic templating

To build it I have used node + express with mongoose ORM and ejs for templating.

2 Routing/Templates

All requests are handled in the file Routes/index.js Only those requests that are defined here will be answered, there is no static file path searching such as in the development server's serve() therefore I don't need to worry about url validation.

All the files delivered are ejs templates stored in views/. All sections that are common to all the pages are stored in views/partials, these are then inserted into the pages in views/pages using for example

```
<% include ../partials/navbar %>
```

this way we keep the html DRY, then if we wish to edit the navbar it can be done in one place.

all pages are delivered using res.render(). Specific data for each page is supplied with this call for example, the navbar needs to know if the user is signed in to display either a sign in or sign out button, so the nav bar contains this line

```
<a href="<%= viewDataSignStatus.signRoute %>" class="navbar-sign-in"> <p><%=: viewDataSignStatus.
    signInOrOut %></p> </a>
```

where ¡%= viewDataSignStatus . signRoute %¿ looks for the viewDataSignStatus . signRoute in the data provided to the render call and substitutes the value. eg

```
    res.render('pages/signin', {
                    viewDataSignStatus: viewDataSignedIn[0],
                    error: errorMessages[response.extras.msg]
                });

//where
viewDataSignedIn = [{signInOrOut: "sign in", signRoute: "./sign-in"},
                        {signInOrOut: "sign out", signRoute: "./sign-out"}],
```

This is also used to display more dynamic layouts e.g. in the index page where we display tiled post previews

Here we supply a array of posts which come back from the DB, each post is presented in this fashion. This again keeps the html dry which was my biggest problem during the client side work.

```
<% previews.forEach(function(post) { %>
                    <div class="content-preview well">
                        <a href="./article?ID=<%= post._id %>"> <h3><%= post.title %></h3></a>
                        <a href="/?author=<%= post.author%>"><mark class="author"><%= post.author%></
    mark></a>
                        <br/>
                        <div class="embed-responsive embed-responsive-16by9">
                        <%- post.media -%>
                         </div>

                        <span class="content-tags">
                            <% post.tags.forEach(function(tag) { %>
                                <a href="/?filter=<%= tag%>"> <%= tag%> </a>
                            <% }); %>
                         </span>
                        <p><a class="btn btn-default btn-expand-article" href="./article?ID=<%= post.
    _id %>" role="button">Read</a></p>
                    </div>
                <% }); %>
```

To present intermittent error messages such as ”passwords don’t match” using ejs I have defined my own custom filter which will supply a default value if no error data attribute is given to the render() call. This stops me having to supply an error : ”” every time when its not needed.

```
//custom ejs filter, sets to default value if data not supplied
ejs.filters.get = function(obj, prop, def) {
  return obj[prop] === undefined ? def : obj[prop];
};

and in the html:

    <p><%=: locals | get:'error','' %> </p>
```

3 User Accounts & Sessions

Users can create an account by going to ’/register’ and filling in the form. To implement this I followed this tutorial: `http://miamicoder.com/2014/using-mongodb-and-mongoose-for-user-registration-login-and-logout-in` using a separate accountController module for the logic and DB calls.

each user is stored in my DB in this schema:

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

var UserSchema = new Schema({
    username: String,
    email: String,
    passwordHash: String,
    passwordSalt: String
});

module.exports = mongoose.model('User', UserSchema);
```

When the registration data comes in it is converted to this schema using

```
AccountController.prototype.getUserFromUserRegistration = function(userRegistrationModel) {
    var me = this;
    if (userRegistrationModel.password !== userRegistrationModel.passwordConfirm) {
        return new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.PASSWORD_CONFIRM_MISMATCH } });
    }

    var passwordSaltIn = this.uuid.v4(),
        cryptoIterations = 10000,
        cryptoKeyLen = 64,
        passwordHashIn;

    var user = new this.User({
        username: userRegistrationModel.username,
        email: userRegistrationModel.email,
        passwordHash: this.crypto.pbkdf2Sync(userRegistrationModel.password, passwordSaltIn, cryptoIterations, cryptoKeyLen).toString('hex'),
        passwordSalt: passwordSaltIn
    });
    return new me.ApiResponse({ success: true, extras: { user: user } });
}
```

the passwords are stored as a hash + salt. The salt is generated using node-uuid which provides ”Cryptographically strong random # generation ” and the hash computed using PBKDF2 encryption.

the returned object is added to the db with

```
AccountController.prototype.register = function (newUser, callback) {
    var me = this;
    me.userModel.findOne({ username: newUser.username }, function (err, user) {
        if (err) {
            return callback(err,new me.ApiResponse({ success: false,extras: { msg: me.ApiMessages.DB_ERROR } }));
        }
        if (user) {
            return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.USERNAME_ALREADY_EXISTS } }));
        } else {
            newUser.save(function (err, user, numberAffected) {
                if (err) {
                    return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.DB_ERROR } }));
                }
                if (numberAffected === 1) {
                    var userProfileModel = new me.UserProfile({
                        email: user.email,
                        username: user.username,
                    });
                    return callback(err, new me.ApiResponse({success: true, extras: {userProfileModel: userProfileModel}}));
                } else {
                    return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.COULD_NOT_CREATE_USER } }));
                }
            });
        }
    });
};
```

Similarly there is a log on method:

```
AccountController.prototype.logon = function(username, password, callback) {
    var me = this;
    me.userModel.findOne({ username: username }, function (err, user) {
        if (err) {
            return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.
    DB_ERROR } }));
        }
        if (user) {
            me.hashPassword(password, user.passwordSalt, function (err, passwordHash) {
                if (passwordHash.toString('hex') === user.passwordHash) {
                    var userProfileModel = new me.UserProfile({
                        email: user.email,
                        username: user.username,
                    });
                    me.session.userProfileModel = userProfileModel;
                    me.session.id = me.uuid.v4();
                    return callback(err, new me.ApiResponse({
                        success: true, extras: {
                            userProfileModel: userProfileModel,
                            sessionId: me.session.id
                        }
                    }));
                } else {
                    return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.
    ApiMessages.INVALID_PWD } }));
                }
            });
        } else {
            return callback(err, new me.ApiResponse({ success: false, extras: { msg: me.ApiMessages.
    USERNAME_NOT_FOUND } }));
        }
    });
};
```

One issue I struggled with for a few days is that the passwords were always failing the check

```
                if (passwordHash == user.passwordHash)
```

even though when printed out they looked identical, and when compared using an online string comparison tool they were shown to match.. in the end I found that mongodb will garble them unless you add .toString('hex') when storing them.

This logon method also stores users sessions using the same crypto random number generator as the secret. The session data is stored in mongo automatically using express-session and mongoose-session packages. This allows sessions to persist through app restarts.

I have decided to follow the trend of all https sites. I realise you said in this case we should have a http server that forwards requests to the https one, but since we have to specify the port number in the url, and the http port must be different it doesn't seem useful to do so for this hosting method.

4 Posts

Once logged in users can upload a post from the form on the /post page.

This is the Schema for a post:

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

var postsSchema = new Schema({
    ID: Schema.Types.ObjectId,
    author: String,
    tags: [String],
    title: String,
    media: String,
    body: String,
    created: { type: Date, default: Date.now },
    comments: [{
            author: String,
            body: String,
            date:  { type: Date, default: Date.now },
    }]
});

module.exports = mongoose.model('Post', postsSchema);
```

Handling these posts was why I used mongo rather than SQL. Storing separate tables of comments and tags and having to join them every time we display a post seemed inefficient/laborious, and knowing that I would never need to compare data between posts it matters little if the data between them is inconsistent.

When adding a post through the route post('/post') I again employ a separate PostController module (controllers/post.js) for building the post model and adding it to the DB which keeps the routing function short. However for other simpler tasks I found that validating the apiResponses needed as much code as just accessing the database there so I choose to do it in the route functions.

Each post is displayed on the /article route with the article ID supplied in a query string. First we find the one article with that ID, then we find other articles by the same author and display previews of those in the ejs forEach() method shown on page 2. We do the same thing for each comment on the main article with the data going into this template in the ejs:

```
 <!-- Comment -->
            <% comments.forEach(function(comment) { %>

              <div class="row">
               <div class=" well col-xs-12 col-sm-10 col-md-8">
                   <div class="media">
                       <div class="media-body">
                           <h4 class="media-heading">
                               <a href="/?author=<%= comment.author%>"><mark class="author"><%=
    comment.author%></mark></a>
                               <small><%=comment.date %></small>
                            </h4>
                           <%=comment.body %>
                       </div>
                   </div>
                   <br/>
               </div>
            </div>

           <% }); %>
```

Since deciding to use ejs, basically because I found a simple looking tutorial, I have read people recommending not to use it, but for this site it has been very effective doing everything I need it to do without any hiccups.

If the user is logged in and viewing their own article a 'Delete' link will be displayed, using the same conditional ejs filter function as the error messages. This links to a /delete with the article ID as a query string which will remove the post from the DB.

Filling in the tags field in the navbar and hitting filter sends you to

```

router.post('/', function(req, res) {
    var tags = req.body.tags.split(",").map(Function.prototype.call, String.prototype.trim);
    if(tags.length==0) {
        res.redirect("/");
    }
    var qString = qs.stringify({filter: tags});
    var url = "/?" + qString;
    res.statusCode = 302;
    res.setHeader("Location", url);
    res.end();
});
```

which adds each of the comma separated tags to a filter= query string and sends that to the main route:

```

/* GET home page. */
router.get('/', function(req, res) {
    var accountController = new AccountController(User, req.session);
    var signedIn = accountController.session.userProfileModel !== undefined ? 1 : 0;
    if(req.query.author !== undefined) {
        var author = req.query.author;
        Post.find().where('author').equals(author).sort({ created: -1 }).limit(10).exec(function(err,
     authorsPosts) {
            if (err) return res.send("error");
            if(authorsPosts.length==0) {
                res.render('pages/index', {
                    viewDataSignStatus: viewDataSignedIn[signedIn],
                    previews: authorsPosts,
                    error: "Sorry there are no posts with that tag."
                });
            } else {
                res.render('pages/index', {
                    viewDataSignStatus: viewDataSignedIn[signedIn],
                    previews: authorsPosts
                });
            }
        });
    } else if(req.query.filter !== undefined) {
        var tagList = req.query.filter.constructor == Array ? req.query.filter : req.query.filter.
    split(",");
        Post.find( { tags : { $elemMatch: { $in : tagList } } } ).limit(10).exec(function(err,
    taggedPosts) {
            if (err) return res.send("error");
            if(taggedPosts.length==0) {
                res.render('pages/index', {
                    viewDataSignStatus: viewDataSignedIn[signedIn],
                    previews: taggedPosts,
                    error: "Sorry there are no posts with that tag."
                });
            } else {
                res.render('pages/index', {
                    viewDataSignStatus: viewDataSignedIn[signedIn],
                    previews: taggedPosts
                });
            }
        });
    } else {
        Post.find().sort({ created: 1 }).limit(10).exec(function(err, latestPosts) {
            if (err) return res.send(err);
            res.render('pages/index', {
                viewDataSignStatus: viewDataSignedIn[signedIn],
                previews: latestPosts
            });
        });
    }
});
```

this also handles query strings of ?author= which users are sent to by clicking the links attached to every authors name. the line

```
var tagList = req.query.filter.constructor == Array ? req.query.filter : req.query.filter. split(",")
    ;
```

took me a while to figure was necessary. Since if a single tag is specified then when we unpack the query string the variable is a single string, but we need it to be an array for the db query to work.

5 Conclusion

Now a list of features I am proud of under each of the marking headings:

5.1 Set up / General Server Issues

- Routing system makes site impregnible to filesystem attacks (I think).
- https certificates and session handled with minimal fuss.

5.2 Database Integration

- database queries integrated seemlessly with dynamic layouts to create flexible auto updating pages.

5.3 Programming

- As modular a design as reasonably possible.
- View system provides as DRY html as possible.

5.4 Sophistication / Ingenuity / Creativity / Polish

- comments system
- multi tag searching navbar.