Easy Node Authentication: Setup and Local

Chris Sevilleja ((https://scotch.io/author/chris)@sevilayha (https://twitter.com/sevilayha)) December 4, 2013 324

Tutorials (https://scotch.io/category/tutorials) authentication (https://scotch.io/tag/authentication), node.js (https://scotch.io/tag/node-js)

This tutorial has been updating for ExpressJS 4.0. Read more **here (/tutorials/javascript/upgrading-our-easy-node-authentication-series-to-expressjs-4-0)**.

Authentication and logins in Node can be a complicated thing. Actually logging in for any application can be a pain. This article series will deal with authenticating in your Node application using the package Passport (http://passportjs.org).

What we'll be building:

We will build an application that will have:

- **Local account logins** and signups (using passport-local (https://github.com/jaredhanson/passport-local))
- **Facebook logins** and registration (using passport-facebook (https://github.com/jaredhanson/passport-facebook))
- **Twitter logins** and registration (using passport-twitter (https://github.com/jaredhanson/passport-twitter))
- **Google+ logins** and registration (using oauth with passport-google-oauth (https://github.com/jaredhanson/passport-google-oauth))
- Require login for certain routes/sections of your application
- Creating a password hash for local accounts (using bcrypt-nodejs (https://github.com/shaneGirish/bcrypt-nodejs))
- Displaying error messages (using flash with connect-flash (https://github.com/jaredhanson/connect-flash). required since express 3.x)
- Linking all social accounts under one user account

- Allowing a user to unlink a specific social account

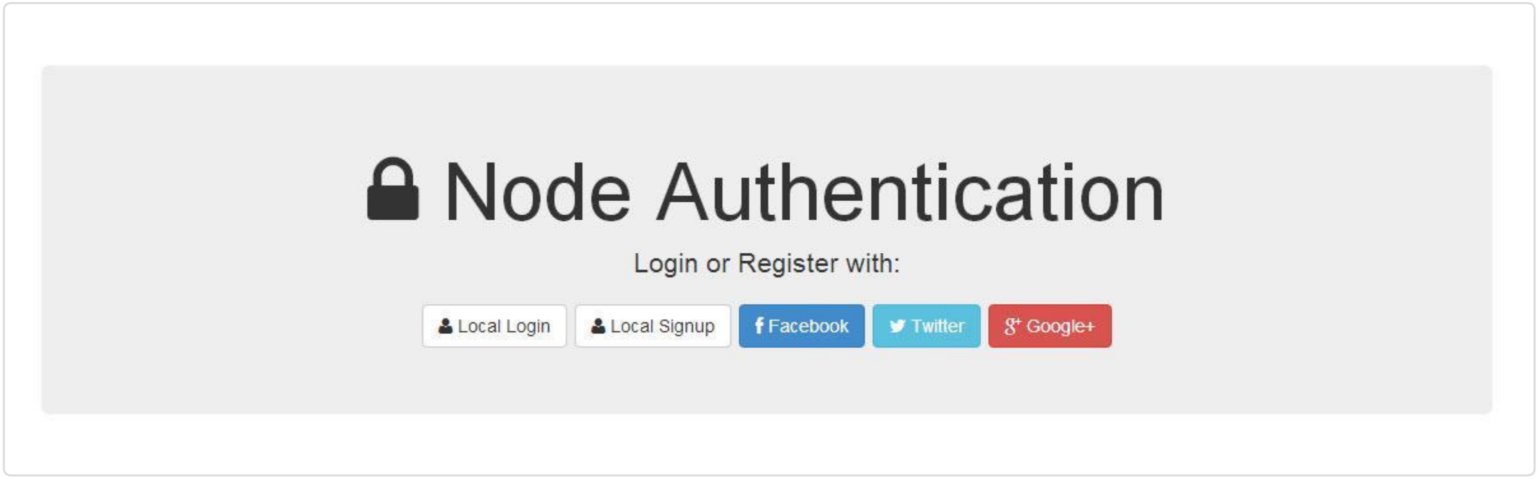
The Tutorial Series

We’ll release this entire gigantic series over 5 different tutorials. They are:

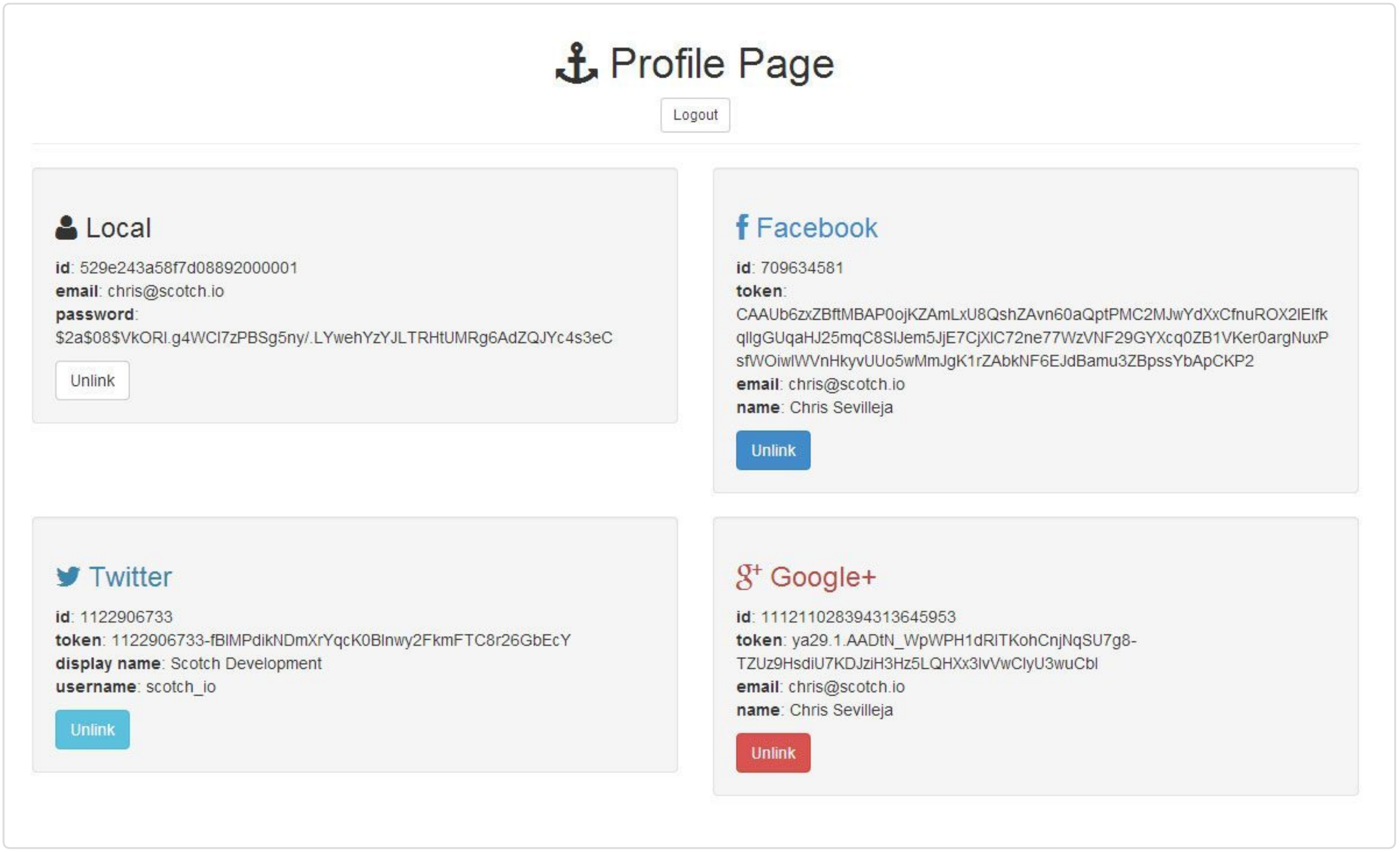
1. Getting Set up and Local Authentication
2. Facebook Authentication
3. Twitter Authentication
4. Google Authentication
5. Linking all accounts together

Enough chit chat. Let’s dive right into a completely blank Node application and build our entire application from scratch.

Here’s what we’ll be building:



(https://cask.scotch.io/2013/12/node-authentication.jpg) And after a user has logged in with all their credentials:



Local

id: 529e243a58f7d08892000001
email: chris@scotch.io
password:
$2a$08$VkORI.g4WCI7zPBSg5ny/.LYwehYzYJLTRHtUMRg6AdZQJYc4s3eC

Facebook

id: 709634581
token:
CAAUb6zxZBftMBAP0ojKZAmLxU8QshZAvn60aQptPMC2MJwYdXxCfnuROX2lElfk
qllgGUqaHJ25mqC8SlJem5JjE7CjXlC72ne77WzVNF29GYXcq0ZB1VKer0argNuxP
sfWOiwlWVnHkyvUUo5wMmJgK1rZAbkNF6EJdBamu3ZBpssYbApCKP2
email: chris@scotch.io
name: Chris Sevilleja

Twitter

id: 1122906733
token: 1122906733-fBIMPdikNDmXrYqcK0BInwy2FkmFTC8r26GbEcY
display name: Scotch Development
username: scotch_io

Google+

id: 111211028394313645953
token: ya29.1.AADtN_WpWPH1dRITKohCnjNqSU7g8-
TZUz9HsdiU7KDJziH3Hz5LQHXx3lvVwClyU3wuCbl
email: chris@scotch.io
name: Chris Sevilleja

(https://cask.scotch.io/2013/12/node-authentication-profile1.jpg) For this article, we'll be focusing on setup and only local logins and registrations/signups. Since this is the first article and also deals with setting up our application, it will probably be one of the longer ones if not the longest. Sit tight for the duration of your flight.

Setting Up Our Application

To set up our base Node application, we'll need a few things. We'll set up our **npm packages**, **node application**, **configuration files**, **models**, and **routes**.

Application Structure

```
- app
------ models
---------- user.js  <!-- our user model -->
------ routes.js    <!-- all the routes for our application -->
- config
------ auth.js      <!-- will hold all our client secret keys (facebook, twitter,
------ database.js  <!-- will hold our database connection settings -->
------ passport.js  <!-- configuring the strategies for passport -->
- views
------ index.ejs    <!-- show our home page with login links -->
------ login.ejs    <!-- show our login form -->
------ signup.ejs   <!-- show our signup form -->
------ profile.ejs  <!-- after a user logs in, they will see their profile -->
- package.json      <!-- handle our npm packages -->
- server.js         <!-- setup our application -->
```

Go ahead and create all those files and folders and we'll fill them in as we go along.

Packages package.json

We are going to install all the packages needed for the entire tutorial series. This means we'll install all the packages needed for passport local, facebook, twitter, google, and the other things we need. It's all commented out so you know what each does.

```
// package.json

{
  "name": "node-authentication",
  "main": "server.js",
  "dependencies" : {
    "express" : "~4.0.0",
    "ejs" : "~0.8.5",
    "mongoose" : "~3.8.1",
    "passport" : "~0.1.17",
    "passport-local" : "~0.1.6",
    "passport-facebook" : "~1.0.2",
    "passport-twitter" : "~1.0.2",
    "passport-google-oauth" : "~0.1.5",
    "connect-flash" : "~0.1.1",
    "bcrypt-nodejs" : "latest",

    "morgan": "~1.0.0",
    "body-parser": "~1.0.0",
    "cookie-parser": "~1.0.0",
    "method-override": "~1.0.0",
    "express-session": "~1.0.0"
  }
}
```

Most of these are pretty self-explanatory.

- *Express* is the framework.
- *Ejs* is the templating engine.
- *Mongoose* is object modeling for our MongoDB database.
- *Passport* stuff will help us authenticating with different methods.
- *Connect-flash* allows for passing session flashdata messages.
- *Bcrypt-nodejs* gives us the ability to hash the password.

I use bcrypt-nodejs instead of bcrypt since it is easier to set up in windows. For more information on the newer ExpressJS 4.0 dependencies (morgan, body-parser, cookie-parser, method-override, express-session), see this article on

ExpressJS 4.0 (https://scotch.io/bar-talk/expressjs-4-0-new-features-and-upgrading-from-3-0#removed-bundled-middleware). Now that we have all of our dependencies ready to go, let’s go ahead and install them:

`npm install` With all of our packages ready to go, let’s set up our application in `server.js`.

Application Setup server.js

Let’s make all our packages work together nicely. Our goal is to set up this file and try to have it bootstrap our entire application. We’d like to not go back into this file if it can be helped. This file will be the glue for our entire application.

```
// server.js

// set up ======================================================================
// get all the tools we need
var express  = require('express');
var app      = express();
var port     = process.env.PORT || 8080;
var mongoose = require('mongoose');
var passport = require('passport');
var flash    = require('connect-flash');

var morgan       = require('morgan');
var cookieParser = require('cookie-parser');
var cookieParser = require('cookie-parser');
var bodyParser   = require('body-parser');
var session      = require('express-session');


var configDB = require('./config/database.js');

// configuration ===============================================================
mongoose.connect(configDB.url); // connect to our database

// require('./config/passport')(passport); // pass passport for configuration

// set up our express application
app.use(morgan('dev')); // log every request to the console
app.use(cookieParser()); // read cookies (needed for auth)
app.use(bodyParser()); // get information from html forms

app.set('view engine', 'ejs'); // set up ejs for templating

// required for passport
app.use(session({ secret: 'ilovescotchscotchyscotchscotch' })); // session secret
app.use(passport.initialize());
app.use(passport.session()); // persistent login sessions
app.use(flash()); // use connect-flash for flash messages stored in session

// routes ======================================================================
require('./app/routes.js')(app, passport); // load our routes and pass in our app and

// launch ======================================================================
app.listen(port);
```

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

We are going to comment out our passport configuration for now. We'll uncomment it after we create that `config/passport.js` file.

The path of our **passport** object is important to note here. We will create it at the very beginning of the file with `var passport = require('passport');`. Then we pass it into our `config/passport.js` file for it to be configured. Then we pass it to the `app/routes.js` file for it to be used in our routes.

Now with this file, we have our application listening on **port 8080**. All we have to do to start up our server is:

`node server.js` Then when we visit **http://localhost:8080** we will see our application. (Not really right this moment since we have some more set up to do)

Auto Refreshing: By default, node doesn't automatically refresh our server every time we change files. To do that we'll use **nodemon (https://github.com/remy/nodemon)**. Just install with: `npm install -g nodemon` and use with: `nodemon server.js`.

Now this won't do much for our application since we don't have our **database configuration**, **routes**, **user model**, or **passport configuration** set up. Let's do the database and routes now.

Database Config config/database.js

We already are calling this file in `server.js`. Now we just have to set it up.

```
// config/database.js
module.exports = {

    'url' : 'your-settings-here' // looks like mongodb://<user>:<pass>@mongo.onmodulu

};
```

Fill this in with your own database. If you don't have a MongoDB database lying around, I would suggest going to Modulus.io (https://modulus.io/) and grabbing one. Once you sign up (and you get a $15 credit for signing up), you can create your database, grab its **connection url**, and place it in this file.

You can also install MongoDB locally and use a local database. You can find instructions here: An Introduction to MongoDB (https://scotch.io/tutorials/an-introduction-to-mongodb).

Routes app/routes.js

We will keep our routes simple for now. We will have the following routes:

- Home Page (/)
- Login Page (/login)
- Signup Page (/signup)
- Handle the POST for both login
- Handle the POST for both signup
- Profile Page (after logged in)

```
// app/routes.js
module.exports = function(app, passport) {

    // =====================================
    // HOME PAGE (with login links) ========
    // =====================================
    app.get('/', function(req, res) {
        res.render('index.ejs'); // load the index.ejs file
    });

    // =====================================
    // LOGIN ===============================
    // =====================================
    // show the login form
    app.get('/login', function(req, res) {

        // render the page and pass in any flash data if it exists
        res.render('login.ejs', { message: req.flash('loginMessage') });
    });

    // process the login form
    // app.post('/login', do all our passport stuff here);

    // =====================================
    // SIGNUP ==============================
    // =====================================
    // show the signup form
    app.get('/signup', function(req, res) {

        // render the page and pass in any flash data if it exists

        res.render('signup.ejs', { message: req.flash('signupMessage') });
    });

    // process the signup form
    // app.post('/signup', do all our passport stuff here);

    // =====================================
    // PROFILE SECTION =====================
    // =====================================
    // we will want this protected so you have to be logged in to visit
    // we will use route middleware to verify this (the isLoggedIn function)
    app.get('/profile', isLoggedIn, function(req, res) {
        res.render('profile.ejs', {
            user : req.user // get the user out of session and pass to template
        });
    });

    // =====================================
    // LOGOUT ==============================
    // =====================================
    app.get('/logout', function(req, res) {
        req.logout();
        res.redirect('/');
    });
};
```

```
// route middleware to make sure a user is logged in
function isLoggedIn(req, res, next) {


    // if user is authenticated in the session, carry on
    if (req.isAuthenticated())
        return next();

    // if they aren't redirect them to the home page
    res.redirect('/');
}
```

app.post: For now, we will comment out the routes for handling the form POST. We do this since passport isn't set up yet.

req.flash: This is the connect-flash way of getting flashdata in the session. We will create the `loginMessage` inside our passport configuration.

isLoggedIn: Using **route middleware**, we can protect the profile section route. A user has to be logged in to access that route. Using the `isLoggedIn` function, we will kick a user back to the home page if they try to access `http://localhost:8080/profile` and they are **not logged in**.

Logout: We will handle logout by using `req.logout()` provided by passport. After logging out, redirect the user to the home page.

With our server running, we can visit our application in our browser at **http://localhost:8080**. Once again, we won't see much since we haven't made our **views**. Let's go do that now. (We're almost to the authentication stuff, I promise).

Views views/index.ejs, views/login.ejs, views/signup.ejs

Here we'll define our views for our **home page**, **login page**, and **signup/registration page**.

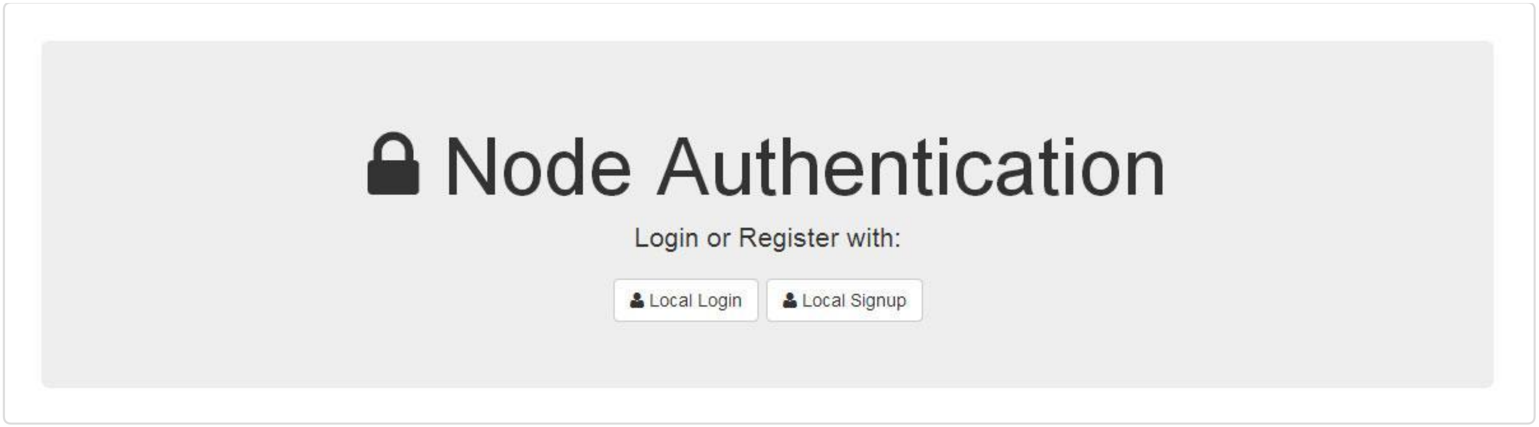
Home Page views/index.ejs

Our home page will just show links to all our forms of authentication.

```
<!-- views/index.ejs -->
<!doctype html>
<html>
```

```
<html>
<head>
    <title>Node Authentication</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootst
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/fon
    <style>
        body        { padding-top:80px; }
    </style>
</head>
<body>
<div class="container">

    <div class="jumbotron text-center">
        <h1><span class="fa fa-lock"></span> Node Authentication</h1>

        <p>Login or Register with:</p>

        <a href="/login" class="btn btn-default"><span class="fa fa-user"></span> Loc
        <a href="/signup" class="btn btn-default"><span class="fa fa-user"></span> Lo
    </div>

</div>
</body>
</html>
```

Now if we visit our app in our browser, we'll have a site that looks like this:



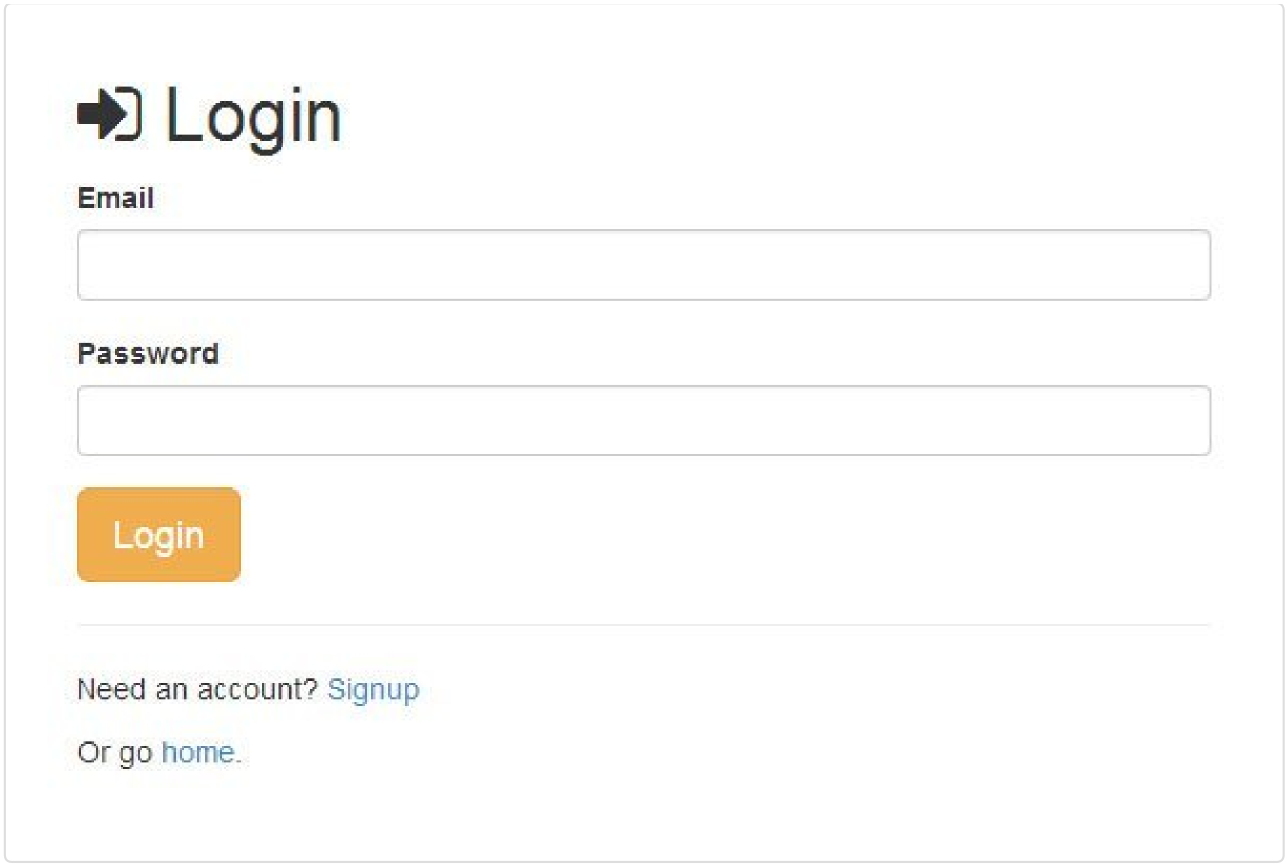
(https://cask.scotch.io/2013/12/node-authentication-local.jpg) Here are the views for our login and signup pages also.

Login Form views/login.ejs

```
<!-- views/login.ejs -->
<!doctype html>
<html>
<head>
    <title>Node Authentication</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootst
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/fon
    <style>
        body        { padding-top:80px; }
    </style>
```

```
</head>
<body>
<div class="container">

<div class="col-sm-6 col-sm-offset-3">

    <h1><span class="fa fa-sign-in"></span> Login</h1>

    <!-- show any messages that come back with authentication -->
    <% if (message.length > 0) { %>
        <div class="alert alert-danger"><%= message %></div>
    <% } %>

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

        <button type="submit" class="btn btn-warning btn-lg">Login</button>
    </form>

    <hr>
```

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


    </div>

    </div>
    </body>
    </html>
```



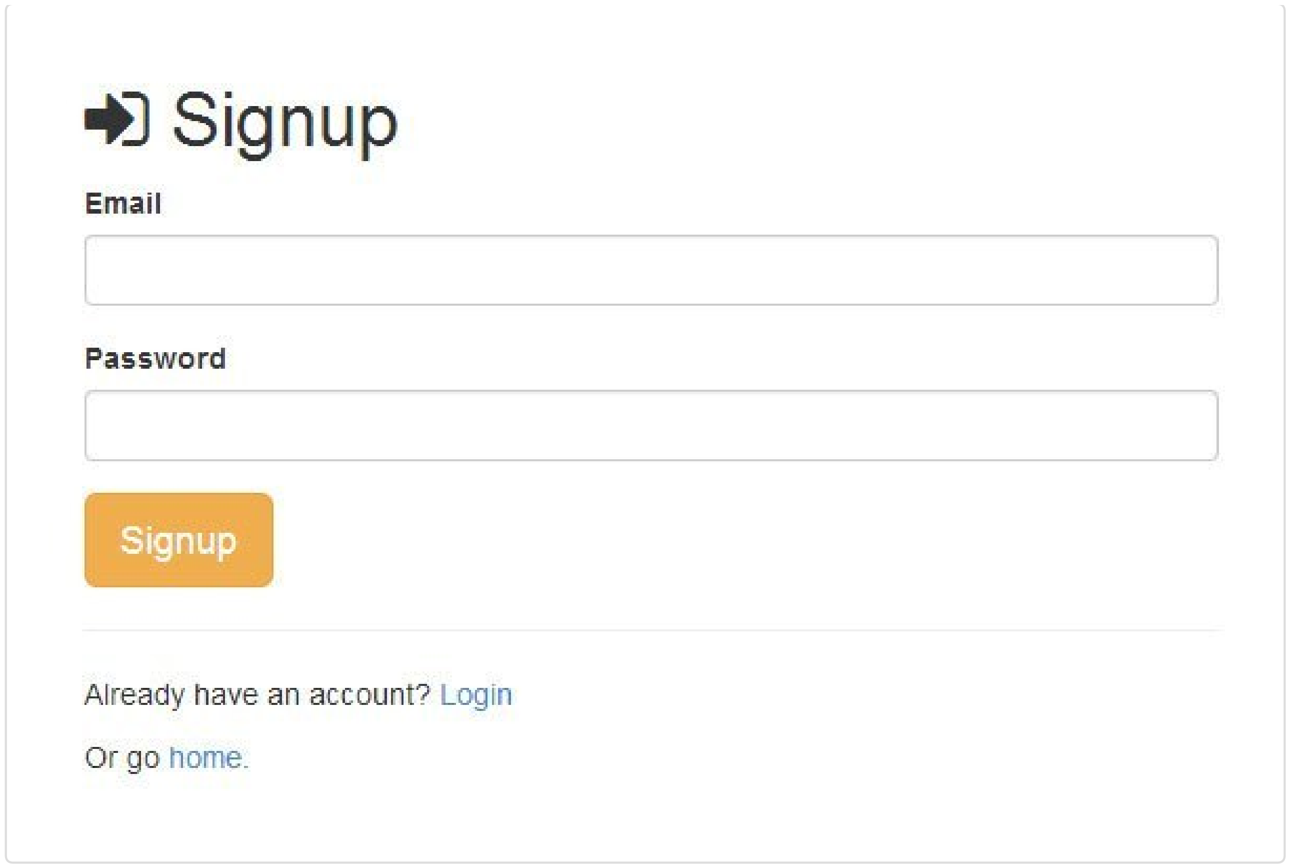
(https://cask.scotch.io/2013/12/node-auth-local-login.jpg)

Signup Form views/signup.ejs

```
<!-- views/signup.ejs -->
<!doctype html>
<html>
```

```
<head>
    <title>Node Authentication</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootst
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/fon
    <style>
        body        { padding-top:80px; }
    </style>
</head>
<body>
<div class="container">

<div class="col-sm-6 col-sm-offset-3">

    <h1><span class="fa fa-sign-in"></span> Signup</h1>

    <!-- show any messages that come back with authentication -->
    <% if (message.length > 0) { %>
        <div class="alert alert-danger"><%= message %></div>
    <% } %>

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

```
                </div>

                <button type="submit" class="btn btn-warning btn-lg">Signup</button>

            </form>

            <hr>

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

        </div>

        </div>
        </body>
        </html>
```



(https://cask.scotch.io/2013/12/node-auth-local-signup.jpg)

Authenticating With Passport Locally

Finally! We have finally set up our application and have gotten to the authentication part. Don't worry. The rest of the authentication articles in this tutorial series will use the same base so we won't have to set up our application again.

So far we have **installed our packages**, **set up our application**, **connected to our database**, **created our routes**, and **created our views**.

Now we will create our **user model**, **configure passport for local authentication**, and use our configured passport to **process our login/signup forms**.

User Model

We will create our user model for the entire tutorial series. Our user will have the ability to be linked to multiple social accounts and to a local account. For local accounts, we will be keeping **email** and **password**. For the social accounts, we will be keeping their **id**, **token**, and some user information.

You can change these fields out to be whatever you want. You can authenticate locally using username and password (passport-local actually uses username by default but we'll change that to email).

```
// app/models/user.js
// load the things we need
var mongoose = require('mongoose');
var bcrypt   = require('bcrypt-nodejs');

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

    local            : {
        email        : String,
        password     : String,
    },
    facebook         : {
        id           : String,
        token        : String,
        email        : String,
        name         : String
    },
    twitter          : {
        id           : String,
        token        : String,
        displayName  : String,
        username     : String
    },
    google           : {
        id           : String,
        token        : String,
        email        : String,
        name         : 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 = mongoose.model('User', userSchema);
```

Our model is done. We will be hashing our password within our user model before it saves to the database. This means we don't have to deal with generating the hash ourselves. It is all handled nicely and neatly inside our user model.

Let’s move onto the important stuff of this article: **authenticating locally**!

Configuring Passport for Local Accounts

All the configuration for passport will be handled in `config/passport.js`. We want to keep this code in its own file away from our other main files like routes or the server file. I have seen some implementations where passport will be configured in random places. I believe having it in this config file will keep your overall application clean and concise.

So far, we created our passport object in `server.js`, and then we pass it to our `config/passport.js` file. This is where we configure our **Strategy** for local, facebook, twitter, and google. This is also the file where we will create the `serializeUser` and `deserializeUser` functions to store our user in session.

I would highly recommend going to read the passport docs (http://passportjs.org/guide/) to understand more about how the package works.

Handling Signup/Registration

We will be handling login and signup in `config/passport.js`. Let’s look at signup first.

```
// config/passport.js

// load all the things we need
var LocalStrategy   = require('passport-local').Strategy;

// load up the user model
var User            = require('../app/models/user');

// 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 wi
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire request to the
    },
    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 al
    } 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);
        });
    }

});
```

```
            });

        }));

};
```

We have now provided a strategy to passport called **local-signup**. We will use this strategy to process our signup form. Let’s open up our `app/routes.js` and handle the POST for our signup form.

```
// app/routes.js
...

    // process the signup form
    app.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
        failureFlash : true // allow flash messages
    }));

...
```

That’s all the code we need for the route. All of the heavy duty stuff lives inside of `config/passport.js`. All we have to set here is where our failures and successes get redirected. Super clean.

There is also much more you can do with this. Instead of specifying a `successRedirect`, you could use a callback and take more control over how your application works. Here is a great stackoverflow answer (http://stackoverflow.com/questions/15711127/express-passport-node-js-error-handling) on error handling. It explains how to use `done()` and how to be more specific with your handling of a route.

Testing Signup

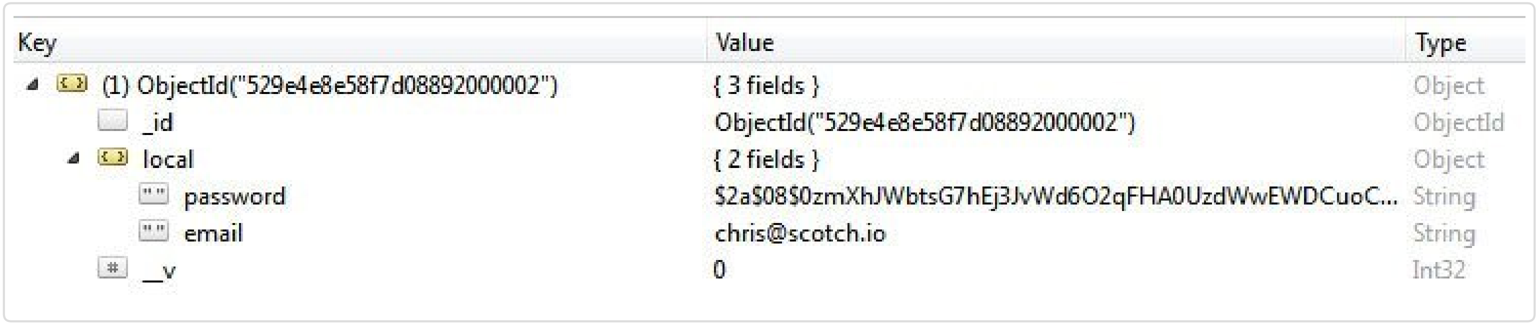
With our passport config finally laid out, we can uncomment that line in our `server.js`. This will load our config and then we can use our signup form.

```
// server.js
...

    // uncomment this line
    require('./config/passport')(passport); // pass passport for configuration

...
```

Now that we have passport, our routes, and our redirects in place, let’s go ahead and test our signup form. In your browser, go to **http://localhost:8080/signup** and fill out your form.

If all goes according to plan, you should be **logged in**, your user **saved in the session**, and you are **redirected to the /profile page** (the profile page will show nothing right now since we haven’t defined that view).

If we look in our database, we’ll also see our user sitting there cozily with all the credentials we created for him.



Key	Value	Type
(1) ObjectId("529e4e8e58f7d08892000002")	{ 3 fields }	Object
_id	ObjectId("529e4e8e58f7d08892000002")	ObjectId
local	{ 2 fields }	Object
password	$2a$08$0zmXhJWbtsG7hEj3JvWd6O2qFHA0UzdWwEWDCuoC...	String
email	chris@scotch.io	String
__v	0	Int32

(https://cask.scotch.io/2013/12/node-user-in-database.jpg)

Exploring Your Database: I use **Robomongo (http://robomongo.org/)**

to see what’s in my database. Just download it and connect to your database to see your new users after they signup!

With users able to sign up, let’s give them a way to login.

Login

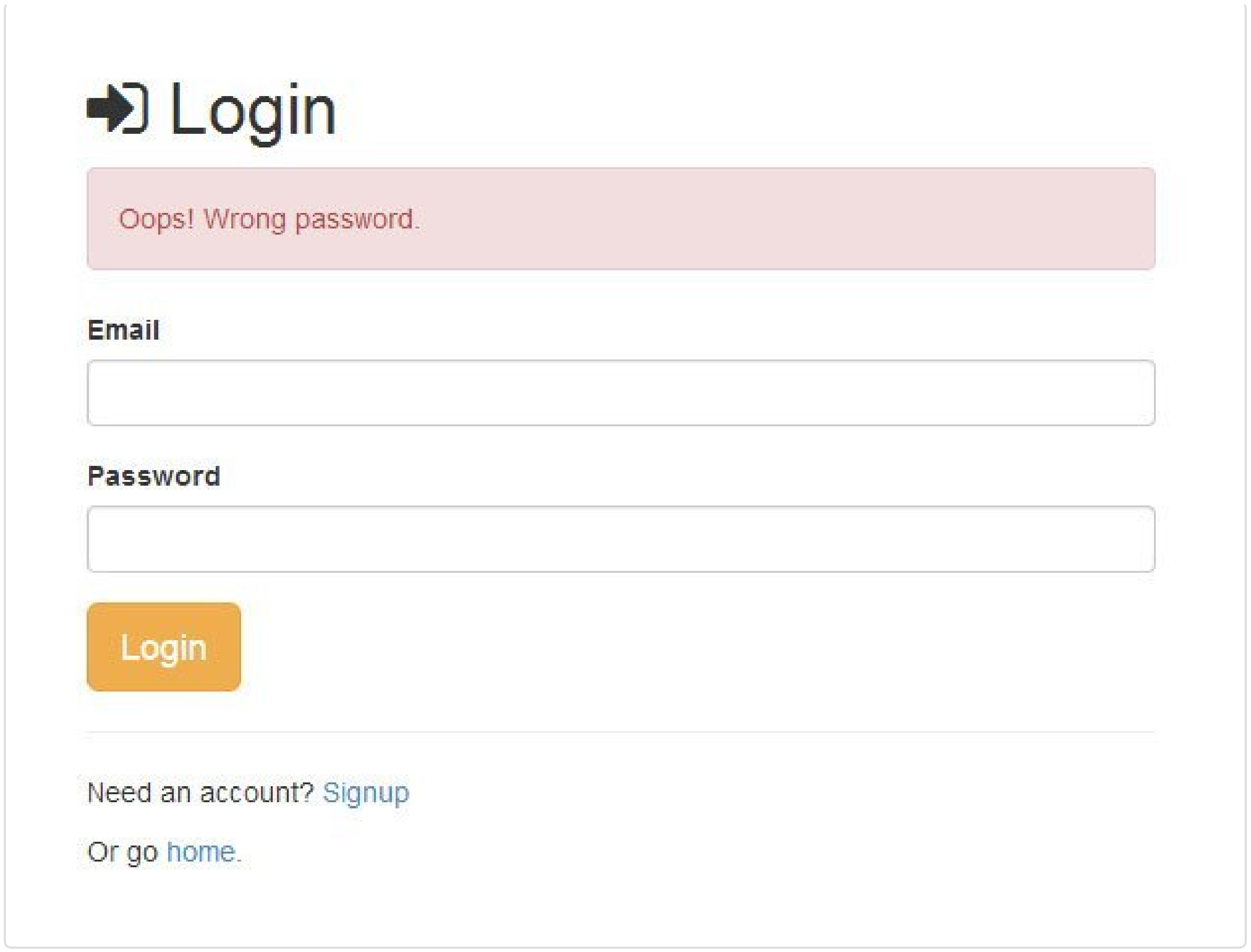
This will be very similar to the signup strategy. We’ll add the strategy to our `config/passport.js` and the route in `app/routes.js`.

```
// config/passport.js

...

    // =========================================================================
    // 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 wi
        usernameField : 'email',
        usernameField : 'email',
        passwordField : 'password',
        passReqToCallback : true // allows us to pass back the entire request to the
    },

    function(req, email, password, done) { // callback with email and password from o

        // 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.'))

            // if the user is found but the password is wrong
            if (!user.validPassword(password))
                return done(null, false, req.flash('loginMessage', 'Oops! Wrong passw

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

    }));

};
```

We have now provided a strategy to passport called **local-login**. We will use this strategy to process our login form. We can check if a user exists, if the password is wrong, and set flash data to show error messages. Let’s open up our `app/routes.js` and handle the POST for our login form.

```
// app/routes.js
...

    // process the login form
    app.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 a
        failureFlash : true // allow flash messages
    }));

...
```

If you try to login with a user email that doesn’t exist in our database, you will see the error. Same goes for if your password is wrong.


Login

Oops! Wrong password.

(https://cask.scotch.io/2013/12/node-auth-login-error.jpg)

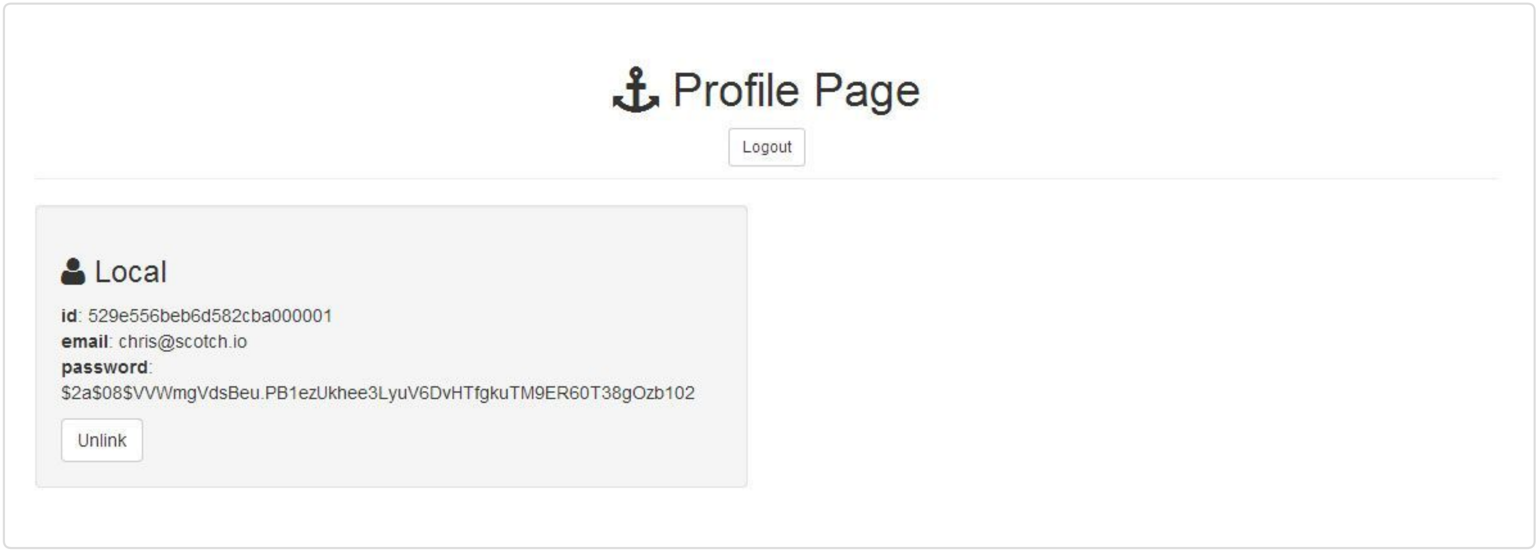
Displaying User and Secure Profile Page

views/profile.ejs

Now we have functional signup and login forms. If a user is successful in authenticating they will be redirected to the profile page. If they are not successful, they will go home. The last thing we need to do is make our profile page so that those that are lucky enough to signup (all of us?) will have an exclusive place of our site all to themselves.

```
<!-- views/profile.ejs -->
<!doctype html>
<html>
<head>
    <title>Node Authentication</title>
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/bootstrap/3.0.2/css/bootst
    <link rel="stylesheet" href="//netdna.bootstrapcdn.com/font-awesome/4.0.3/css/fon
    <style>
        body        { padding-top:80px; word-wrap:break-word; }
    </style>
</head>
<body>
<div class="container">

    <div class="page-header text-center">
        <h1><span class="fa fa-anchor"></span> Profile Page</h1>
        <a href="/logout" class="btn btn-default btn-sm">Logout</a>
    </div>
    <div class="row">

        <!-- LOCAL INFORMATION -->

        <div class="col-sm-6">
            <div class="well">
                <h3><span class="fa fa-user"></span> Local</h3>

                    <p>
                        <strong>id</strong>: <%= user._id %><br>
                        <strong>email</strong>: <%= user.local.email %><br>
                        <strong>password</strong>: <%= user.local.password %>
                    </p>

            </div>
        </div>

    </div>

</div>
</body>
</html>
```

After a user logs in, they can see all their information. It is grabbed from the session and passed to our view in the `app/routes.js` file. We will also provide a link to logout.



id: 529e556beb6d582cba000001
email: chris@scotch.io
password:
$2a$08$VVWmgVdsBeu.PB1ezUkhee3LyuV6DvHTfgkuTM9ER60T38gOzb102

(https://cask.scotch.io/2013/12/node-auth-local-profile-page.jpg)

Conclusion

There you have it! We've built a brand new application from scratch and have the ability to let users signup/register and login. We even have support for flash messages, hashing passwords, and requiring login for some sections of our site using route middleware.

Coming up next we'll be looking at how to take this same structure, and use passport to authenticate with Facebook, Twitter, and Google. After that we'll look at how we can get all these thing working together in the same application. Users will be able to login with one type of account, and then link their other accounts.

As always, if you see any ways to improve this or need any clarification, sound off in the comments and we'll respond pretty close to immediately… pretty close.

Edit #1: Changed password hashing to be handled inside user model and asynchronously. **Edit #2**: Changed password hashing to be explicitly called. Helps with future tutorials.

This article is part of our **Easy Node Authentication (https://scotch.io/series/easy-node-authentication)** series.

- Getting Started and Local Authentication

- **Facebook (https://scotch.io/tutorials/javascript/easy-node-authentication-facebook)**
- **Twitter (https://scotch.io/tutorials/javascript/easy-node-authentication-twitter)**
- **Google (https://scotch.io/tutorials/javascript/easy-node-authentication-google)**
- **Linking All Accounts Together (https://scotch.io/tutorials/javascript/easy-node-authentication-linking-all-accounts-together)**
- **Upgrading for ExpressJS 4.0 (/tutorials/javascript/upgrading-our-easy-node-authentication-series-to-expressjs-4-0)**