**User Authentication with the MEAN Stack**

By Simon Holmes February 01, 2016

*https://www.sitepoint.com/user-authentication-mean-stack/*

In this article we’re going to look at managing user authentication in the MEAN stack. We’ll use the most common MEAN architecture of having an Angular single page app using a REST API built with Node, Express and MongoDB.

When thinking about user authentication, we need to tackle the following things:

Let a user register

Save their data, but never directly store their password

Let a returning user log in

Keep a logged in user’s session alive between page visits

Have some pages that can only been seen by logged in users

Change output to the screen depending on logged in status (e.g. a “login” button or a “my profile” button)

Before we dive into the code, let’s take a few minutes for a high level look at how authentication is going to work in the MEAN stack.

The MEAN Stack Authentication Flow

So what does authentication look like in the MEAN stack?

Still keeping this at a high level, these are the **components of the flow**:

* User data is stored in MongoDB, with the passwords hashed
* *CRUD functions are built in an Express API* – Create (register), Read (login, get profile), Update, Delete
* An Angular application calls the API and deals with the responses
* The Express API generates a *JSON Web Token* (JWT, pronounced “Jot”) upon registration or login, and passes this to the Angular application
* The Angular application stores the JWT in order to maintain the user’s session
* The Angular application checks the validity of the JWT when displaying protected views
* The Angular application passes the JWT back to Express when calling protected API routes

JWTs are preferred over cookies for maintaining the session state in the browser; cookies are better for maintaining state when using a server-side application.

**The Example Application**

The code for this article is available on GitHub. To run the application, you will need to have Node.js installed, along with MongoDB (For instructions on how to install, please refer to Mongo’s official documentation — Windows, Linux, MacOS).

The Angular App

To keep the example in this article simple, we’ll start with an Angular SPA with four pages:

Homepage

Register page

Login page

Profile page

The pages are pretty basic and look like this to start with:

Screenshots of the app

The profile page will only be accessible to authenticated users. All of the files for the Angular app are in a folder inside the Express app called app\_client, and is arranged like this:

Screenshot of app\_client folder structure

There is a Gulp process to concatenate all of the JavaScript files into app.min.js.

The REST API

We’ll also start off with the skeleton of a REST API built with Node, Express and MongoDB, using Mongoose to manage the schemas. This API has three routes:

/api/register (POST) – to handle new users registering

/api/login (POST) – to handle returning users logging in

/api/profile/USERID (GET) – to return profile details when given a USERID

The code for the API is all held in another folder inside the Express app, called app\_api. This holds the routes, controllers and model, and is organized like this:

Screenshot of app\_api folder structure

At this starting point each of the controllers simply responds with a confirmation, like this:

module.exports.register = function(req, res) {

console.log("Registering user: " + req.body.email);

res.status(200);

res.json({

"message" : "User registered: " + req.body.email

});

};

Okay, let’s get on with the code, starting with the database.

Creating the MongoDB Data Schema with Mongoose

There’s a simple user schema defined in /app\_api/models/users.js. It defines the need for an email address, a name, a hash and a salt – the hash and salt will be used instead of saving a password. The email is set to unique as we’ll use it for the login credentials. Here’s the schema.

var userSchema = new mongoose.Schema({

email: {

type: String,

unique: true,

required: true

},

name: {

type: String,

required: true

},

hash: String,

salt: String

});

Managing the Password without Saving It

Saving user passwords is a big no-no. Should a hacker get a copy of your database you want to make sure that they can’t use it to log in to accounts. This is where the hash and salt come in.

The salt is a string of characters unique to each user. The hash is created by combining the password provided by the user and the salt, and then applying one-way encryption. As the hash cannot be decrypted, the only way to authenticate a user is to take the password, combine it with the salt and encrypt it again. If the output of this matches the hash, then the password must have been correct.

To do the setting and the checking of the password we can use Mongoose schema methods – these are essentially functions that you add to the schema. These will both make use of the Node.js crypto module.

At the top of the users.js model file, require crypto so that we can use it:

var crypto = require('crypto');

Nothing needs installing as crypto ships as part of Node. Crypto itself has several methods; we’re interested in randomBytes to create the random salt and pbkdf2Sync to create the hash (there’s much more about Crypto in the Node.js API docs).

Setting the Password

To save the reference to the password we can create a new method called setPassword on the userSchema schema that accepts a password parameter. The method will then use crypto.randomBytes to set the salt, and crypto.pbkdf2Sync to set the hash.

userSchema.methods.setPassword = function(password){

this.salt = crypto.randomBytes(16).toString('hex');

this.hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');

};

We’ll use this method when creating a user; instead of saving the password to a password path we will be able to pass it to the setPassword function to set the salt and hash paths in the user document.

Checking the Password

Checking the password is a similar process, but we already have the salt from the Mongoose model. This time we just want to encrypt the salt and the password and see if the output matches the stored hash.

Add another new method to the users.js model file, called validPassword.

userSchema.methods.validPassword = function(password) {

var hash = crypto.pbkdf2Sync(password, this.salt, 1000, 64).toString('hex');

return this.hash === hash;

};

Generating a JSON Web Token (JWT)

One more thing that the Mongoose model needs to be able to do is generate a JWT, so that the API can send it out as a response. A Mongoose method is ideal here too, as it means we can keep the code in one place and call it whenever needed – we’ll need to call it when a user registers and when a user logs in.

To create the JWT we’ll use a module called jsonwebtoken which needs to be installed in the application, so run this on the command line:

npm install jsonwebtoken --save

And then require this in the users.js model file:

var jwt = require('jsonwebtoken');

This module exposes a sign method that we can use to create a JWT, simply passing it the data we want to include in the token, plus a secret that the hashing algorithm will use. The data should be sent as a JavaScript object, and include an expiry date in an exp property.

Adding a generateJwt method to userSchema in order to return a JWT looks like this:

userSchema.methods.generateJwt = function() {

var expiry = new Date();

expiry.setDate(expiry.getDate() + 7);

return jwt.sign({

\_id: this.\_id,

email: this.email,

name: this.name,

exp: parseInt(expiry.getTime() / 1000),

}, "MY\_SECRET"); // DO NOT KEEP YOUR SECRET IN THE CODE!

};

Note: It is important that your secret is kept safe – only the originating server should know what it is. It is best practice to set the secret as an environment variable, and not have it in the source code, especially if your code is stored in version control somewhere.

That’s everything we need to do with the database.

Setup Passport to Handle the Express Authentication

Passport is a Node module that simplifies the process of handling authentication in Express. It provides a common gateway to work with many different authentication “strategies”, such as logging in with Facebook, Twitter or Oauth. The strategy we’ll use is called “local”, as it uses a username and password stored locally.

To use Passport, first install it and the strategy, saving them in package.json.

npm install passport --save

npm install passport-local --save

Configure Passport

Inside the app\_api folder create a new folder config and create a file in there called passport.js. This is where we define the strategy.

Before defining the strategy this file needs to require Passport, the strategy, Mongoose and the User model.

var passport = require('passport');

var LocalStrategy = require('passport-local').Strategy;

var mongoose = require('mongoose');

var User = mongoose.model('User');

For a local strategy we essentially just need to write a Mongoose query on the User model. This query should find a user with the email address specified, and then call the validPassword method to see if the hashes match. Pretty simple.

There’s just one curiosity of Passport to deal with. Internally the local strategy for Passport expects two pieces of data called username and password. However we’re using email as our unique identifier, not username. This can be configured in an options object with a usernameField property in the strategy definition. After that, it’s over to the Mongoose query.

So all in, the strategy definition will look lke this:

passport.use(new LocalStrategy({

usernameField: 'email'

},

function(username, password, done) {

User.findOne({ email: username }, function (err, user) {

if (err) { return done(err); }

// Return if user not found in database

if (!user) {

return done(null, false, {

message: 'User not found'

});

}

// Return if password is wrong

if (!user.validPassword(password)) {

return done(null, false, {

message: 'Password is wrong'

});

}

// If credentials are correct, return the user object

return done(null, user);

});

}

));

Note how the validPassword schema method is called directly on the user instance.

Now Passport just needs to be added to the application. So in app.js we need to require the Passport module, require the Passport config and initialise Passport as middleware. The placement of all of these items inside app.js is quite important, as they need fit into a certain sequence.

The Passport module should be required at the top of the file with the other general require statements.

var express = require('express');

var path = require('path');

var favicon = require('serve-favicon');

var logger = require('morgan');

var cookieParser = require('cookie-parser');

var bodyParser = require('body-parser');

var passport = require('passport');

The config should be required after the model is required, as the config references the model.

require('./app\_api/models/db');

require('./app\_api/config/passport');

Finally Passport should be initialised as Express middleware just before the API routes are added, as these routes are the first time that Passport will be used.

app.use(passport.initialize());

app.use('/api', routesApi);

We’ve now got the schema and Passport set up, next it’s time to put these to use in the routes and controllers of the API.

Configure API End Points

With the API we’ve got two things to do:

Make the controllers functional

Secure the /api/profile route so that only authenticated users can access it

Code the Register and Login API Controllers

In the example app the register and login controllers are in /app\_api/controllers/authentication.js. In order for the controllers to work the file needs to require Passport, Mongoose and the user model.

var passport = require('passport');

var mongoose = require('mongoose');

var User = mongoose.model('User');

The Register API Controller

The register controller needs to do the following:

take the data from the submitted form and create a new Mongoose model instance

Call the setPassword method we created earlier to add the salt and the hash to the instance

Save the instance as a record to the database

Generate a JWT

Send the JWT inside the JSON response

In code, all that looks like this:

module.exports.register = function(req, res) {

var user = new User();

user.name = req.body.name;

user.email = req.body.email;

user.setPassword(req.body.password);

user.save(function(err) {

var token;

token = user.generateJwt();

res.status(200);

res.json({

"token" : token

});

});

};

This makes use of the setPassword and generateJwt methods we created in the Mongoose schema definition. See how having that code in the schema makes this controller really easy to read and understand.

Don’t forget that in reality this code would have a number of error traps, validating form inputs and catching errors in the save function. They are omitted here to highlight the main functionality of the code.

The Login API Controller

The login controller hands over pretty much all control to Passport, although you could (and should) add some validation beforehand to check that the required fields have been sent.

For Passport to do its magic and run the strategy defined in the config we need to call the authenticate method as shown below. This method will call a callback with three possible parameters err, user and info. If user is defined then it can be used to generate a JWT to be returned to the browser.

module.exports.login = function(req, res) {

passport.authenticate('local', function(err, user, info){

var token;

// If Passport throws/catches an error

if (err) {

res.status(404).json(err);

return;

}

// If a user is found

if(user){

token = user.generateJwt();

res.status(200);

res.json({

"token" : token

});

} else {

// If user is not found

res.status(401).json(info);

}

})(req, res);

};

Securing an API Route

The final thing to do in the back-end is make sure that only authenticated users can access the /api/profile route. The way to validate a request is to ensure that the JWT sent with it is genuine, by using the secret again – this is why you should keep it a secret and not in the code.

Configuring the Route Authentication

First we need to install a piece of middleware called express-jwt.

npm install express-jwt --save

Then we need to require it and configure it in the file where the routes are defined; in the sample application this is /app\_api/routes/index.js. Configuration is a case of telling it the secret, and – optionally – the name of the property to create on the req object that will hold the JWT. We’ll be able to use this property inside the controller associated to the route. The default name for the property is user, but this is the name of an instance of our Mongoose User model, so we’ll set it to payload to avoid confusion.

var jwt = require('express-jwt');

var auth = jwt({

secret: 'MY\_SECRET',

userProperty: 'payload'

});

Again, don’t keep the secret in the code!

Applying the Route Authentication

To apply this middleware simply reference the function in the middle of the route to be protected, like this:

router.get('/profile', auth, ctrlProfile.profileRead);

If someone tries to access that route now without a valid JWT the middleware will throw an error. To make sure our API plays nicely, catch this error and return a 401 response by adding the following into the error handlers section of the main app.js file.

// error handlers

// Catch unauthorised errors

app.use(function (err, req, res, next) {

if (err.name === 'UnauthorizedError') {

res.status(401);

res.json({"message" : err.name + ": " + err.message});

}

});

Using the Route Authentication

In this example we only want people to be able to view their own profiles, so we get the user ID from the JWT and use it in a Mongoose query.

The controller for this route is in /app\_api/controllers/profile.js – the entire contents of this file look like this:

var mongoose = require('mongoose');

var User = mongoose.model('User');

module.exports.profileRead = function(req, res) {

// If no user ID exists in the JWT return a 401

if (!req.payload.\_id) {

res.status(401).json({

"message" : "UnauthorizedError: private profile"

});

} else {

// Otherwise continue

User

.findById(req.payload.\_id)

.exec(function(err, user) {

res.status(200).json(user);

});

}

};

Naturally, this should be fleshed out with some more error trapping – for example if the user isn’t found – but this snippet is kept brief to demonstrate the key points of the approach.

That’s it for the back-end. The database is configured, we have API end points for registering and logging in that generate and return a JWT, and also a protected route. On to the front-end!

Create Angular Authentication Service

Most of the work in the front-end can be put into an Angular service, creating methods to manage:

Saving the JWT in local storage

Reading the JWT from local storage

Deleting the JWT from local storage

Calling the register and login API end points

Checking whether a user is currently logged in

Getting the details of the logged in user from the JWT

We’ll need to create a new service called authentication, and register it with the meanApp application. In the example app this is in a file authentication.service.js in the folder /app\_client/common/services.

Local Storage: Saving, Reading and Deleting a JWT

To keep a user logged in between visits we use localStorage in the browser to save the JWT. An alternative is to use sessionStorage, which will only keep the token during the current browser session.

To use localStorage we need to pass the Angular $window service into our service. Then we can create and return three methods saveToken, getToken and logout as shown below.

(function () {

angular

.module('meanApp')

.service('authentication', authentication);

authentication.$inject = ['$http', '$window'];

function authentication ($http, $window) {

var saveToken = function (token) {

$window.localStorage['mean-token'] = token;

};

var getToken = function () {

return $window.localStorage['mean-token'];

};

logout = function() {

$window.localStorage.removeItem('mean-token');

};

return {

saveToken : saveToken,

getToken : getToken,

logout : logout

};

}

})();

From the perspective of the service, logging out is simply a case of deleting the JWT from local storage. Now let’s add a method to check for this token – and the validity of the token – to find out if the visitor is logged in.

Check Whether a User Is Logged In

Add a new method called isLoggedIn to the service, and reference it in the return statement of course.

This method will need to call the getToken method to try and read the mean-token from local storage. If the token exists, the method will also need to validate that the JWT has not expired.

Getting Data from a JWT

When we set the data for the JWT (in the generateJwt Mongoose method) we included the expiry date in an exp property. But if you look at a JWT it seems to be a random string, like this following example.

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJfaWQiOiI1NWQ0MjNjMTUxMzcxMmNkMzE3YTRkYTciLCJlbWFpbCI6InNpbW9uQGZ1bGxzdGFja3RyYWluaW5nLmNvbSIsIm5hbWUiOiJTaW1vbiBIb2xtZXMiLCJleHAiOjE0NDA1NzA5NDUsImlhdCI6MTQzOTk2NjE0NX0.jS50GlmolxLoKrA\_24LDKaW3vNaY94Y9EqYAFvsTiLg

So how do you read a JWT?

A JWT is actually made up of three separate strings, separated by a dot .. These three parts are:

Header – An encoded JSON object containing the type and the hashing algorithm used

Payload – An encoded JSON object containing the data, the real body of the token

Signature – An encrypted hash of the header and payload, using the “secret” set on the server

It’s the second part we’re interested in here, the payload. Note that this is encoded rather than encrypted, meaning that we can decode it.

There’s a function called atob() that is native to modern browsers, that will decode a Base64 string like this (sorry IE9 users, there is a polyfill for you though).

So we need to get the second part of the token, decode it and parse it as JSON. Then we can check that the expiry date hasn’t passed.

At the end of it, the isLoggedIn function should return a boolean value, depending on whether a valid token is found or not. Put together, it looks like this:

var isLoggedIn = function() {

var token = getToken();

var payload;

if(token){

payload = token.split('.')[1];

payload = $window.atob(payload);

payload = JSON.parse(payload);

return payload.exp > Date.now() / 1000;

} else {

return false;

}

};

Next we’ll use a similar approach to get the details of the logged in user.

Get the Current User’s Details

To get the user data from the JWT is also a simple process. First validate that the user is logged in, and then get the data you want from the payload.

The new currentUser method looks like this:

var currentUser = function() {

if(isLoggedIn()){

var token = getToken();

var payload = token.split('.')[1];

payload = $window.atob(payload);

payload = JSON.parse(payload);

return {

email : payload.email,

name : payload.name

};

}

};

This will extract the email and name from the JWT and return them inside an object ready to be used.

Calling the Register and Login API End-Points

Just two methods to add. We’ll need an interface between the Angular app and the API, to call the login and register end-points and save the returned token. This will use the Angular $http service – remember to inject this into the authentication service if it’s not already there.

Both of these methods should accept an object, which will contain the details or credentials.

register = function(user) {

return $http.post('/api/register', user).success(function(data){

saveToken(data.token);

});

};

login = function(user) {

return $http.post('/api/login', user).success(function(data) {

saveToken(data.token);

});

};

That finalises the service, now to tie everything together in the Angular app.

Apply Authentication to Angular App

We can use the authentication services inside the Angular app in a number of ways to give the experience we’re after.

Wire up the register and sign in forms

Update the navigation to reflect the user’s status

Only allow logged in users access to the /profile route

Call the protected /api/profile API route

Connect the Register and Login Controllers

We’ll begin by looking at the register and login forms.

The Register Page

The HTML for the registration form already exists and has ng-model attributes attached to the fields, all in the namespance vm.credentials. The form will also have an ng-submit handler. In the example application is it in /app\_client/auth/register/register.view.html and looks like this:

<form ng-submit="vm.onSubmit()">

<div class="form-group">

<label for="name">Full name</label>

<input type="text" class="form-control" id="name" placeholder="Enter your name" ng-model="vm.credentials.name">

</div>

<div class="form-group">

<label for="email">Email address</label>

<input type="email" class="form-control" id="email" placeholder="Enter email" ng-model="vm.credentials.email">

</div>

<div class="form-group">

<label for="password">Password</label>

<input type="password" class="form-control" id="password" placeholder="Password" ng-model="vm.credentials.password">

</div>

<button type="submit" class="btn btn-default">Register!</button>

</form>

The first task in the controller is to ensure our authentication service is injected and available. Next, inside the onSubmit handler for the function, call authentication.register passing it the credentials from the form.

The register method reutrns a $http call, which in turn returns a promise. So we can use that to listen for completion – either an error or success. If someone has successfully registered we’ll set the application to redirect them to the profile page.

In the sample application the controller is in /app\_client/auth/register/register.controller.js and looks like this:

function registerCtrl($location, authentication) {

var vm = this;

vm.credentials = {

name : "",

email : "",

password : ""

};

vm.onSubmit = function () {

authentication

.register(vm.credentials)

.error(function(err){

alert(err);

})

.then(function(){

$location.path('profile');

});

};

The Login Page

The login page is very similar in nature to the register page, but in this form we don’t ask for the name, just email and password. In the sample application it’s in /app\_client/auth/login/login.view.html and looks like this:

<form ng-submit="vm.onSubmit()">

<div class="form-group">

<label for="email">Email address</label>

<input type="email" class="form-control" id="email" placeholder="Enter email" ng-model="vm.credentials.email">

</div>

<div class="form-group">

<label for="password">Password</label>

<input type="password" class="form-control" id="password" placeholder="Password" ng-model="vm.credentials.password">

</div>

<button type="submit" class="btn btn-default">Sign in!</button>

</form>

Once again we have the form submit handler, and model attributes for each of the inputs. In the controller we want the same functionality as the register controller, but this time called the login method of the authentication service.

In the sample application the controller is in /app\_client/auth/login/login.controller.js and look like this.

function loginCtrl($location, authentication) {

var vm = this;

vm.credentials = {

email : "",

password : ""

};

vm.onSubmit = function () {

authentication

.login(vm.credentials)

.error(function(err){

alert(err);

})

.then(function(){

$location.path('profile');

});

};

}

Now users can register and sign in to the application. Note that again there should be more validation in the forms to ensure that all required fields are filled before submitting; these examples are kept to the bare minimum to highlight the main functionality.

Change Content Based on User Status

In the navigation we want to show the Sign in link if a user isn’t logged in, and their username with a link to the profile page if they are logged in.

First we’ll look at the controller. We simply want to call the isLoggedIn and currentUser methods from the navigation controller, and save the return values in the view model. In the sample app the file is in /app\_client/common/directives/navigation/navigation.controller.js and looks like this:

function navigationCtrl($location, authentication) {

var vm = this;

vm.isLoggedIn = authentication.isLoggedIn();

vm.currentUser = authentication.currentUser();

}

That’s pretty simple right? Now in the associated view template we can use ng-hide and ng-show based on the value of navvm.isLoggedIn to display either the sign in link or the profile link. To add the user’s name to the profile link we can access the name property of navvm.currentUser – remember that this is getting the data from the JWT.

In the sample app the file is in /app\_client/common/directives/navigation/navigation.view.html and the updated part looks like this:

<ul class="nav navbar-nav navbar-right">

<li ng-hide="navvm.isLoggedIn"><a href="login">Sign in</a></li>

<li ng-show="navvm.isLoggedIn"><a href="profile">{{ navvm.currentUser.name }}</a></li>

</ul>

Protect a Route for Logged in Users Only

In this step we’ll see how to make a route accessible only to logged in users, by protecting the /profile path.

To do this we need to check whenever the route changes in the Angular app by using a watcher on $routeChangeStart. When a new route is selected we need to check whether it is the profile page, and whether the user is logged in. If it is the profile page and the user is not logged in, then we’ll redirect to the homepage.

The best place to put this is in a run module added to the Angular application. This module will need access to the following services:

$rootScope to listen for the $routeChangeStart event

$location to see what the new path is

authentication to use our service, calling the isLoggedIn method

This run function should be added to the main Angular setup file in /app\_client/main.js, which should now look like this:

(function () {

angular.module('meanApp', ['ngRoute']);

function config ($routeProvider, $locationProvider) {

$routeProvider

.when('/', {

templateUrl: 'home/home.view.html',

controller: 'homeCtrl',

controllerAs: 'vm'

})

.when('/register', {

templateUrl: '/auth/register/register.view.html',

controller: 'registerCtrl',

controllerAs: 'vm'

})

.when('/login', {

templateUrl: '/auth/login/login.view.html',

controller: 'loginCtrl',

controllerAs: 'vm'

})

.when('/profile', {

templateUrl: '/profile/profile.view.html',

controller: 'profileCtrl',

controllerAs: 'vm'

})

.otherwise({redirectTo: '/'});

// use the HTML5 History API

$locationProvider.html5Mode(true);

}

function run($rootScope, $location, authentication) {

$rootScope.$on('$routeChangeStart', function(event, nextRoute, currentRoute) {

if ($location.path() === '/profile' && !authentication.isLoggedIn()) {

$location.path('/');

}

});

}

angular

.module('meanApp')

.config(['$routeProvider', '$locationProvider', config])

.run(['$rootScope', '$location', 'authentication', run]);

})();

With that run function in place, now if an unauthenticated user tries to visit the profile page they will be redirected to the homepage.

Call a Protected API Route

The /api/profile route has been set up to check for a JWT in the request, otherwise it will return a 401 unauthorised error.

The application needs a data service, which in the sample app you can find in /app\_client/common/services/data.service.js. This service needs access to our authentication service, so that it can call the getToken method.

To pass the token to the API it needs to be sent through as a header on the request, called Authorization. The following snippet shows the main data service function, and the format required to send the token.

function meanData ($http, authentication) {

var getProfile = function () {

return $http.get('/api/profile', {

headers: {

Authorization: 'Bearer '+ authentication.getToken()

}

});

};

return {

getProfile : getProfile

};

}

Remember that the back-end code is validating that the token is genuine when the request is made, by using the secret known only to the issuing server.

To make use of this in the profile page we just need to update the controller, in /app\_client/profile/profile.controller.js in the sample app. This will populate a vm.user object when the api returns some data.

function profileCtrl($location, meanData) {

var vm = this;

vm.user = {};

meanData.getProfile()

.success(function(data) {

vm.user = data;

})

.error(function (e) {

console.log(e);

});

}

Then of course it’s just a case of updating the bindings in the view (/app\_client/profile/profile.view.html).

<form class="form-horizontal">

<div class="form-group">

<label class="col-sm-3 control-label">Full name</label>

<p class="form-control-static">{{ vm.user.name }}</p>

</div>

<div class="form-group">

<label class="col-sm-3 control-label">Email</label>

<p class="form-control-static">{{ vm.user.email }}</p>

</div>

</form>

And here’s the final profile page, when logged in.

Screenshot of the profile page

That’s how to manage authentication in the MEAN stack, from securing API routes and managing user details to working with JWTs and protecting SPA routes. If you’ve implemented an authentication system like this in one of your own apps and have any tips, tricks, or advice, be sure to share them in the comments below!

Was this helpful?

More:

angular, authentication, Express, mongo, Mongoose, node, node.js, passport, REST

Meet the author

Simon Holmes

Simon has been a full stack developer since the late 1990's, and is a massive fan of good JavaScript. He is the author of "Getting MEAN" and "Mongoose for Application Development", and co-founder of Full Stack Training Ltd.

Toptal matches you with top developers who are guaranteed to succeed.

AD

Latest Themes

Browse all 9 themes

Premium Theme

SitePoint Themes

SitePoint WordPress Restaurant Theme

Premium Theme

SitePoint Themes

SitePoint WordPress Ecommerce Theme

Premium Theme

SitePoint Themes

SitePoint WordPress Portfolio Theme

Latest Courses

Browse all 18 courses

44m

Premium Course

Zsolt Nagy

Introducing TypeScript

1h 3m

Premium Course

Esteban Herrera

MS Bots

3h 7m

Premium Course

M. David Green

Functional JavaScript Programming

Latest Books

Browse all 15 books

Premium Book

James Hibbard

ECMAScript 2015: A SitePoint Anthology

Premium Book

Shaumik Daityari

Jump Start Git

Premium Book

Colin Ihrig

Full Stack JavaScript Development with MEAN

Recommended

1

jQuery's JSONP Explained with Examples

2

Social Logins with Oauth.io - Log in with Anything, Anywhere

3

Phpseclib: Securely Communicating with Remote Servers via PHP

4

How to Properly Deploy Web Apps via SFTP with Git

5

How to Build and Structure a Node.js MVC Application

Sponsors

Toptal Toptal matches you with top developers who are guaranteed to succeed.

Know when and why code breaks Users finding bugs? Searching logs for errors? Find + fix broken code fast!

New and Awesome Tool: See How Your Visitors are Really Using Your Site

Themes Store

Themes Store

Download our new SitePoint Base Theme. 100% FREE. GPL 2.0.

SitePoint Themes, Dec 07

00:10:24

JavaScript

MS Bots in Azure

Esteban Herrera, Feb 22

Get the latest in JavaScript, once a week, for free.

About

Our Story

Advertise

Press Room

Reference

Terms of Use

Privacy Policy

FAQ

Contact Us

Contribute

Visit

SitePoint Home

Themes

Podcast

Forums

Newsletters

Premium

References

Versioning

Connect

© 2000 – 2017 SitePoint Pty. Ltd.

x

See DocumentationeSignature API. The Best API Support You'll Never Need.