Token-Based Authentication for AngularJS and Laravel Apps (continued)

 Ryan Chenkie

30th June, 2015

This post is a continuation of the Token-Based Authentication for AngularJS and Laravel Apps tutorial on Scotch.io. If you haven't read it yet, head over there and take a look before continuing with this one.

Where We Left Off

In the tutorial on Scotch.io we created a new app called jot-bot to look at how to

implement token-based authentication in AngularJS and Laravel by using jwt-auth and Satellizer together. On the Laravel side, jwt-auth let's us generate JSON web tokens when the user inputs their credentials. The token can then be saved in local storage on the client-side with Satellizer where it is accessed and sent with each subsequent request to the API. We protected our API with the middleware that comes with jwt-auth so that the request gets denied if no token is present.

There were a few things for a complete authentication solution that we didn't get to in the last tutorial, including:

- Setting the logged-in user's data (such as name and email address) and their authentication status in local storage or on `$rootScope` so that we can pass their information around from state to state
- A way to redirect the user to the login page if they become logged out somehow (for example, if the token expires)
- How to log the user out and the implications of token-based authentication on logout

We'll look at how to handle all of the above in this tutorial.

GET THE CODE ON GITHUB

SEE THE DEMO SITE

Adding the Authenticated User Route

The first thing we'll need to do is add a new method to our

`AuthenticateController` on the Laravel side so that we can have an object of the currently authenticated user's data returned to us. Where exactly you put this method, whether it be in an existing controller or an entirely new one, is up to you. For the sake of simplicity, we'll stick with our current controller.

```
// app/Http/Controllers/AuthenticateController.php

...

public function getAuthenticatedUser()
    {
        try {

            if (! $user = JWTAuth::parseToken()->authenticate()) {
                return response()->json(['user_not_found'], 404);
            }

        } catch (Tymon\JWTAuth\Exceptions\TokenExpiredException $e) {

            return response()->json(['token_expired'], $e->getStatusCode

        } catch (Tymon\JWTAuth\Exceptions\TokenInvalidException $e) {

            return response()->json(['token_invalid'], $e->getStatusCode

        } catch (Tymon\JWTAuth\Exceptions\JWTException $e) {

            return response()->json(['token_absent'], $e->getStatusCode(

        }

        // the token is valid and we have found the user via the sub cla
        return response()->json(compact('user'));
    }

...
```

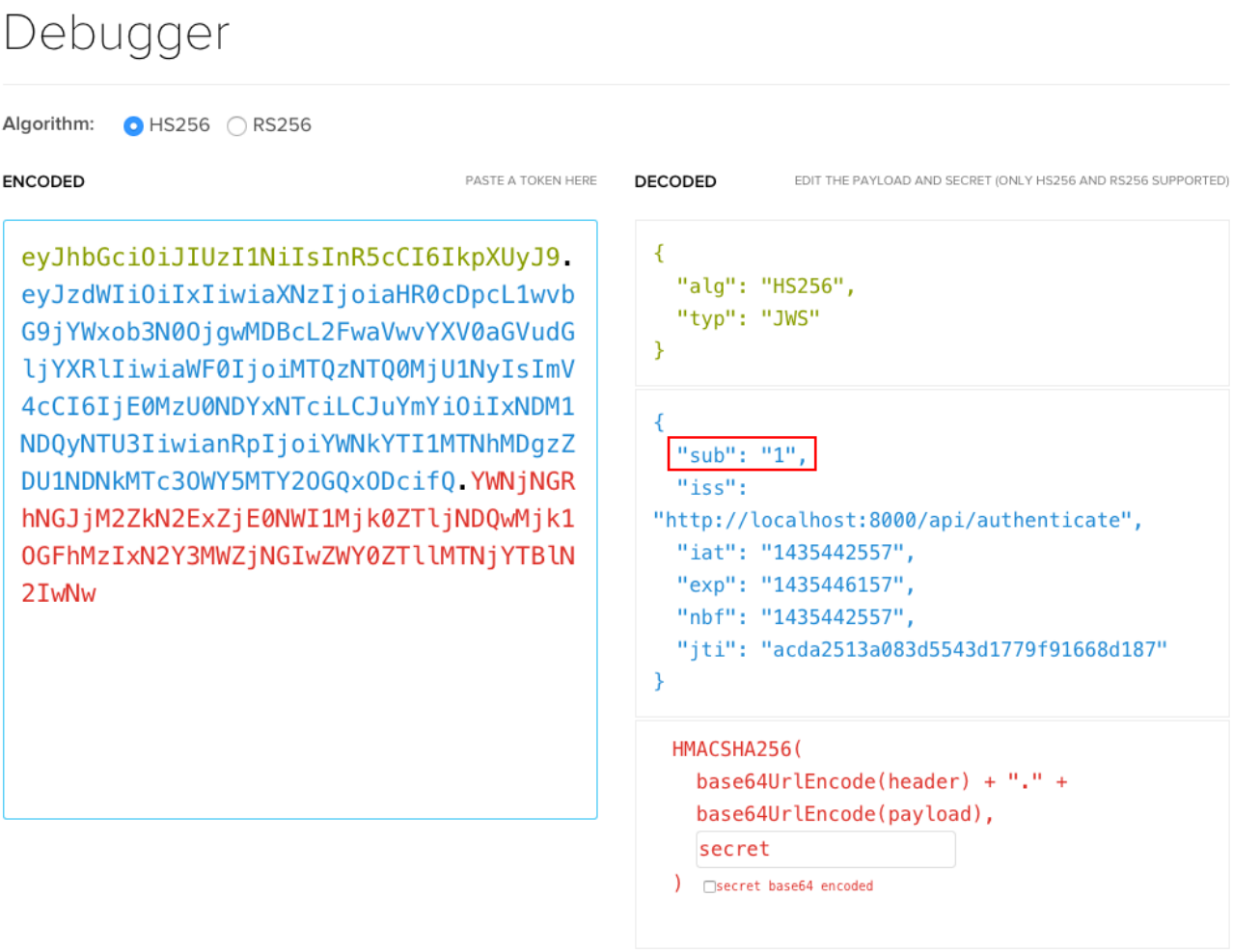
The `getAuthenticatedUser` method grabs the JWT that will be passed along

in the header of a request for the authenticated user. As we'll see later on, when we hit the route that calls this method, we will need to have already generated a token for our user. You'll remember from the first part of the tutorial that generating a token relies on the `authenticate` method within the same controller.

With the JWT passed along with the request, this method is going to use the `JWTAuth` facade to attempt to parse the token and authenticate the user based on it. If the credentials don't match what is in the database, a `404` will be returned along with a message that the user wasn't found.

If authentication was successful, we move onto some more checks using the exception handler that jwt-auth provides, and respond appropriately to each condition. If everything checks out, we return the user that the token belongs to.

You might be wondering how exactly jwt-auth knows which user to authenticate. The answer is that there is a "sub" claim included with the token payload that corresponds to the `id` of the user it belongs to. We can inspect this with the awesome JWT debugger provided by Auth0.



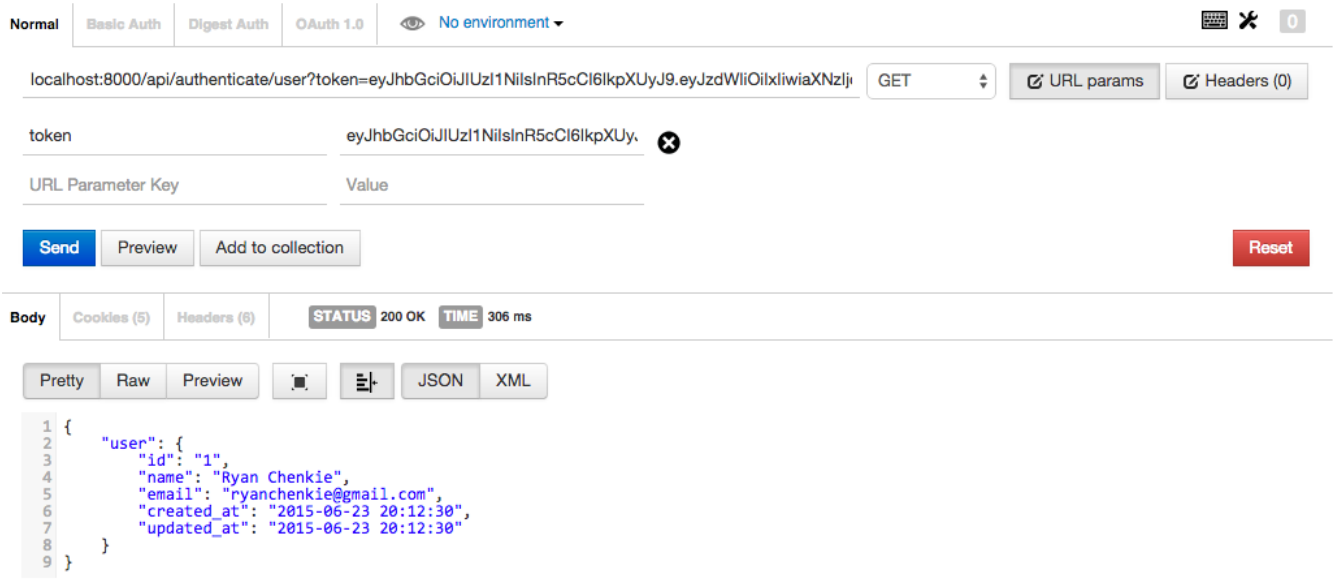
We'll also need to add a new route to handle this method in `routes.php`.

```
// app/Http/routes.php

...

Route::group(['prefix' => 'api'], function()
{
    Route::resource('authenticate', 'AuthenticateController', ['only' =>
    Route::post('authenticate', 'AuthenticateController@authenticate');
    Route::get('authenticate/user', 'AuthenticateController@getAuthentica
});

...
```

We specify that we want a new route at `authenticate/user` that responds to a `GET` request and uses the `getAuthenticatedUser` method on the `AuthenticateController`.

Now that we have the route and controller method in place, we should be able to get our user data returned. Let's test it out using Postman.

> Note: You might need to generate a new token if it's been some time since you generated the last one.



Sending a `GET` request to the `authenticate/user` route of our API with the JWT as a URL parameter let's us successfully retrieve the user data.

Requesting the Authenticated User's Data

Now that we have our API successfully returning the user data, we'll need to setup our authentication controller on the front-end to make a request for the data. As

you'll remember from the first part, the `vm.login` method on our `AuthController` uses Satellizer to make a request to the API for the JWT and then saves it in local storage so it can be sent along with subsequent requests. While we could make a request for the newly authenticated user's data at various times and places in the application, it makes most sense to do so right after we know the Satellizer request for the token was successful. To accomplish this, let's use `$http` to make a `GET` request for the user data in the `then` block of the initial request.

```
// public/scripts/authController.js

(function() {

    'use strict';

    angular
        .module('authApp')
        .controller('AuthController', AuthController);


    function AuthController($auth, $state, $http, $rootScope) {

        var vm = this;

        vm.loginError = false;
        vm.loginErrorText;

        vm.login = function() {

            var credentials = {
                email: vm.email,
                password: vm.password
            }

            $auth.login(credentials).then(function() {

                // Return an $http request for the now authenticated
                // user so that we can flatten the promise chain
```

```
                return $http.get('api/authenticate/user');

            // Handle errors
            }, function(error) {
                vm.loginError = true;
                vm.loginErrorText = error.data.error;

            // Because we returned the $http.get request in the $auth.log
            // promise, we can chain the next promise to the end here
            }).then(function(response) {

                // Stringify the returned data to prepare it
                // to go into local storage
                var user = JSON.stringify(response.data.user);

                // Set the stringified user data into local storage
                localStorage.setItem('user', user);

                // The user's authenticated state gets flipped to
                // true so we can now show parts of the UI that rely
                // on the user being logged in
                $rootScope.authenticated = true;

                // Putting the user's data on $rootScope allows
                // us to access it anywhere across the app
                $rootScope.currentUser = response.data.user;

                // Everything worked out so we can now redirect to
                // the users state to view the data
                $state.go('users');
            });
        }
    }

})();
```

You'll see here that we are now returning the `$http.get` request in the success handler of the `$auth.login` request. While we could take care of the whole `$http.get` request directly in this success handler, if we instead return it then

we can tack another `then` onto the end of the `$auth.login` request which helps to flatten the promise chain. Since we are just dealing with two promises here, it wouldn't be all that bad if we nested them, but in case we ever needed more, this approach helps to keep things cleaner as nesting a lot of promises can start to get messy.

In the `then` block of the request for our user data we are grabbing the response and saving it into local storage. Note that we need to call `JSON.stringify` on the returned object because local storage items need to be saved as text.

Next, we're putting an `authenticated` property on `$rootScope` which is a boolean to let us know that the user has logged in. This is for convenience sake, as we'll now be able to use this `$rootScope` property to conditionally show or hide elements in the view. Likewise, we are saving the user data on `$rootScope` to conveniently access it across the app. Whether or not you use `$rootScope` in this fashion is your call—some prefer not to use it in this manner.

Finally, we are redirecting the user to the `users` state like we were before.

Displaying the User's Name in the View

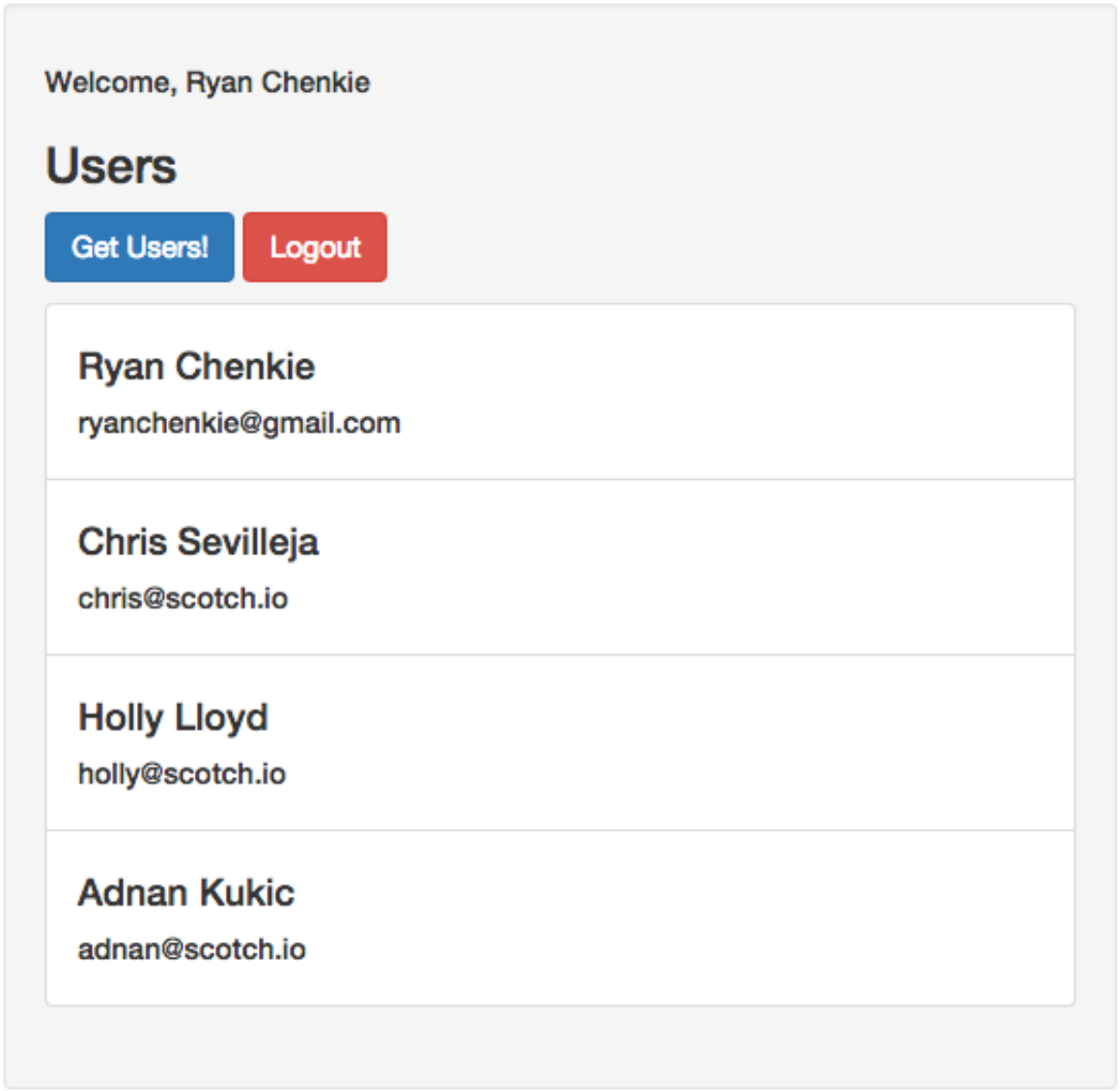
Now that we have the authenticated user's data on `$rootScope` to work with, let's have their name be displayed in the view.

```
<!-- public/views/userView.html -->

<div class="col-sm-6 col-sm-offset-3">
    <div class="well">
        <h5 ng-if="authenticated">Welcome, {{currentUser.name}}</h5>
        <h3>Users</h3>
        <div class="buttons" style="margin-bottom: 10px">
```

```
            <button class="btn btn-primary" ng-click="user.getUsers()">G
            <button class="btn btn-danger" ng-click="user.logout()">Logo
        </div>
        <ul class="list-group" ng-if="user.users">
            <li class="list-group-item" ng-repeat="user in user.users">
                <h4>{{user.name}}</h4>
                <h5>{{user.email}}</h5>
            </li>
        </ul>
        <div class="alert alert-danger" ng-if="user.error">
            <strong>There was an error: </strong> {{user.error.error}}
            <br>Please go back and login again
        </div>
    </div>
</div>
```

You'll see here that we've added an `h5` tag to the view that gets included in the DOM if `$rootScope.authenticated` is `true`. Also on `$rootScope` is the `currentUser` object from which we want the `name` property for the welcome message.

Now when we are at the `users` state we will see the authenticated user's name displayed.



Adding a Logout Method

You'll probably have noticed above that we've added another button to the `users` view which is meant to log the user out. Its `ng-click` is pointing to a `logout` method on `UserController` which we'll create now.

Note: Normally we would want to have all methods dealing with authentication such as login and logout in the same spot. The ideal would be to extract their functionality into a service. For this quick example, we'll put the `logout` method

right into the `UserController` so that we have access to it from the view in our current setup.

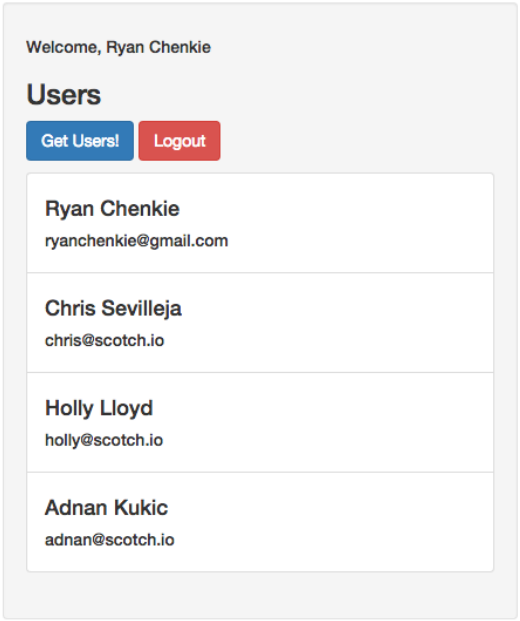
```
// public/scripts/userController.js

(function() {

    'use strict';

    angular
        .module('authApp')
        .controller('UserController', UserController);

    function UserController($http, $auth, $rootScope) {

        var vm = this;

        vm.users;
        vm.error;

        vm.getUsers = function() {

            //Grab the list of users from the API
            $http.get('api/authenticate').success(function(users) {
                vm.users = users;
            }).error(function(error) {
                vm.error = error;
            });
        }

        // We would normally put the logout method in the same
        // spot as the login method, ideally extracted out into
        // a service. For this simpler example we'll leave it here
        vm.logout = function() {

            $auth.logout().then(function() {

                // Remove the authenticated user from local storage
                localStorage.removeItem('user');

                // Flip authenticated to false so that we no longer
```

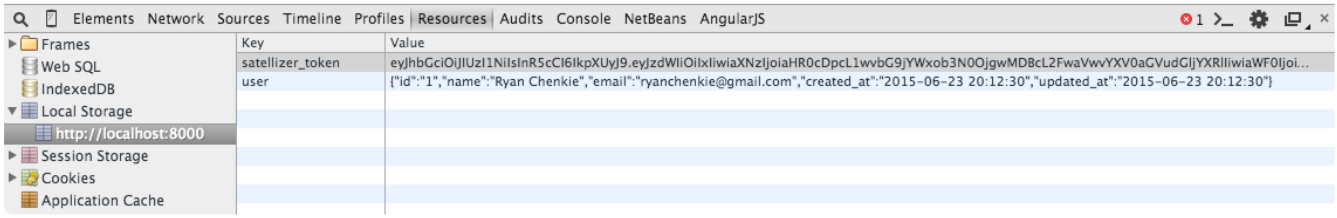
```
                // show UI elements dependant on the user being logged i
                $rootScope.authenticated = false;

                // Remove the current user info from rootscope
                $rootScope.currentUser = null;
            });
        }
    }

})();
```

The `logout` method is going to make use of the `$auth` service provided by Satellizer, much like the `login` method. In the `$auth.logout` success handler we remove the `user` item from local storage and set `$rootScope.authenticated` to false. We also want to remove the `user` data from `$rootScope` as well.

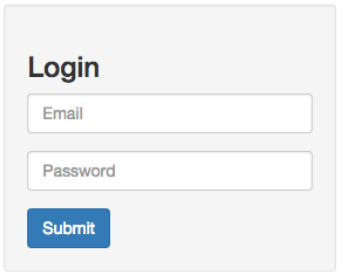
The Satellizer `$auth.logout` method will remove the `satellizer_token` from local storage as well. Let's try logging out to confirm everything is working.

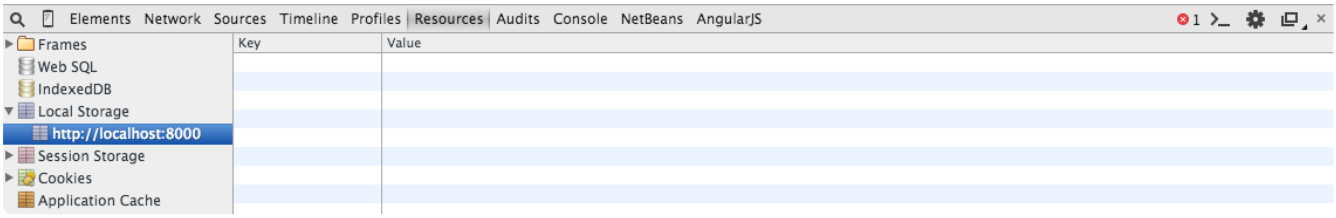
Before logout





After logout





You'll notice that we were redirected to the `auth` state when we logged out. Satellizer handles redirects for us when we use the `$auth.logout` and the default is to take us to the main `/` route. Since our `app.js` file uses `$urlRouterProvider` to specify we want to go to the `auth` route/state anytime a state other than `users` is requested, that's where we're sent.

A Note About JWT's and Logging Out

Now that the user is logged out, further requests to the API will not work because we no longer have a JWT in local storage to be sent along with requests. However, the token is still valid on the Laravel side and could theoretically be used by someone else if they somehow got access to it. There's no effective way to invalidate the JWT, but we can increase our protection by giving all JWT's a short

lifespan. The time to live for JWTs defaults to one hour and can be adjusted in the jwt-auth configuration.

```
// config/jwt.php

...

    /*
    |--------------------------------------------------------------------------
    | JWT time to live
    |--------------------------------------------------------------------------
    |
    | Specify the length of time (in minutes) that the token will be val
    | Defaults to 1 hour
    |
    */

   'ttl' => 60,

...
```

Redirecting the User When Logged Out

The `$auth.logout` method we put in above does a fine job of redirecting us to the `auth` state when we request to be logged out, but as it stands we could still reach the `users` state even though we're not authenticated. We won't be able to see any data because—as we saw in the first part of the tutorial—we get an error message returned by the API when we don't have a JWT to send. Although our API and data are protected, we really should prevent the user from reaching any state other than `auth` if they aren't actually authenticated. We should also redirect the user to the `auth` state once an API error related to a missing or invalid token is encountered. An example of where this would be useful is if the user leaves their browser open and is away from their computer for more than an hour. After

returning they won't be able to use the application because their token will be invalid.

Angular's `$http` service gives us the ability to catch HTTP requests from anywhere in the app by using `interceptors`. Like the AngularJS docs say, `interceptors` are useful for pre-processing requests or post-processing responses. In our case, we're going to want to post-process responses, and specifically, those responses that indicate that we have a missing or invalid token.

To make use of `interceptors` we need to set one up in the config block of `app.js` and then push it onto the `$httpProvider.interceptors` array.

```
c/scripts/app.js

n() {

 strict';

lar
.module('authApp', ['ui.router', 'satellizer'])
.config(function($stateProvider, $urlRouterProvider, $authProvider, $httpP

    function redirectWhenLoggedOut($q, $injector) {

        return {

            responseError: function(rejection) {

                // Need to use $injector.get to bring in $state or else we
                // a circular dependency error
                var $state = $injector.get('$state');

                // Instead of checking for a status code of 400 which migh
                // for other reasons in Laravel, we check for the specific
                // reasons to tell us if we need to redirect to the login
                var rejectionReasons = ['token_not_provided', 'token_expir
```

```
                // Loop through each rejection reason and redirect to the
                // state if one is encountered
                angular.forEach(rejectionReasons, function(value, key) {

                    if(rejection.data.error === value) {

                        // If we get a rejection corresponding to one of t
                        // in our array, we know we need to authenticate t
                        // we can remove the current user from local stora
                        localStorage.removeItem('user');

                        // Send the user to the auth state so they can log
                        $state.go('auth');
                    }
                });

                return $q.reject(rejection);
            }
        }
    }

    // Setup for the $httpInterceptor
    $provide.factory('redirectWhenLoggedOut', redirectWhenLoggedOut);

    // Push the new factory onto the $http interceptor array
    $httpProvider.interceptors.push('redirectWhenLoggedOut');

    $authProvider.loginUrl = '/api/authenticate';
```

We're injecting some new dependencies into the config block— `$httpProvider` and `$provider`. The first thing we do is setup a new function called `redirectWhenLoggedOut` that will contain the logic of what to do when certain response messages are encountered. The `$http.interceptors` array is going to need an object with some specific keys on it to work. In our case we only need to worry about response errors so we specify a key called `responseErrors` on an object that is returned from this function.

We're going to need to make use of the `$state` service to do redirection, but as you'll see, we're bringing it in somewhat differently. If we were to inject it in the traditional way—between the parentheses in the function definition—we'll get a "circular dependency error". Instead, we can use `$injector.get` to inject it in and put it on a variable called `$state` which we can then use to call the actual methods of the `$state` service.

The anonymous function on the `responseError` key has a `rejection` parameter which we can use to grab the data and status associated with the response. As you might have seen in the console when we've made requests without a valid JWT, a `400 bad request` is what gets returned. If we wanted to keep things simple, we could listen for all `400` status codes and redirect to the login page when they are encountered. However, there may be other reasons that a `400` error gets returned from the API that we don't know about. To play it safe and to be very specific to the jwt-auth exceptions, we can also respond the specific status messages that are returned. To handle this, we create an array of `rejectionReasons` that we'll loop through.

Using `angular.forEach`, we loop through each of the `rejectionReasons` and if the rejection error is equal to one of them, we remove the user data from local storage and redirect to the `auth` state.

The `$provide.factory` let's us specify the name of the `interceptor` we want to create and the second argument references the function above that we use to handle the logic for it. We then `push` this on the `$http.interceptors` array.

Initializing the User When the App Loads

So far the way our app handles setting the user state and data on the front-end is rather specific. We set `$rootScope.authenticated` to `true` and `$rootScope.currentUser` to the user's data when the login screen is followed, but what happens if the user doesn't arrive to the app via the `auth` state? If their JWT is expired or otherwise invalid then they will need to go to the login screen anyway, but if they still have a valid token when coming back to the app after navigating away or closing the screen, these properties on `$rootScope` that we're using in the view won't be set.

To fix this we can write some logic that will check whether there is a `user` key set in local storage and if there is, set the `$rootScope` properties mentioned above appropriately. To do so, we'll make use of Angular's `$on` event listener with the `$stateChangeStart` event provided by UI Router. We'll put this all within the `run` block in `app.js`.

```
// public/scripts/app.js

...

.run(function($rootScope, $state) {

            // $stateChangeStart is fired whenever the state changes. We
            // such as toState to hook into details about the state as i
            $rootScope.$on('$stateChangeStart', function(e    '  toState)

                // Grab the user from local storage and parse it to an ol
                var user = JSON.parse(localStorage.getItem('user'));

                // If there is any user data in local storage then the u
                // likely authenticated. If their token is expired, or i
                // otherwise not actually authenticated, they will be re
                // the auth state because of the rejected request anyway
                if(user) {

                    // The user's authenticated state gets flipped to
```

126
Shares

```
                    // true so we can now show parts of the UI that rely
                    // on the user being logged in
                    $rootScope.authenticated = true;

                    // Putting the user's data on $rootScope allows
                    // us to access it anywhere across the app. Here
                    // we are grabbing what is in local storage
                    $rootScope.currentUser = user;

                    // If the user is logged in and we have the auth route
                    // to stay there and can send them to the main state
                    if(toState.name === "auth") {

                        // Preventing the default behavior allows us to use
                        // to change states
                        event.preventDefault();

                        // go to the "main" state which in our case is users
                        $state.go('users');
                    }
                }
            });
        });

...
```

We're using the `$stateChangeStart` event to listen for changes to the state the application is at. The event is fired anytime we move from one state to another and will be fired when we load the application for the first time as well. In the callback we look for a `user` key in local storage and use `JSON.parse` to turn it from a string into an object. If we have a `user` key then we go ahead and set `$rootScope.authenticated` to true and also set `$rootScope.currentUser` to the user data object kept in local storage.

We know that if the user is authenticated then they don't need to see the `auth` state. If they end up there somehow we should really be redirecting them

somewhere else and in this case we'll consider that somewhere else to be the `users` state. We can access the name of the state that we're currently on with `toState.name` and here we check to see if it is equal to `auth`. If it is, we need to redirect to the `users` state with `$state.go`. The event's default behavior will prevent us from changing states so we need to prevent the default behavior for it to work.

With this logic setup in the `$stateChangeStart` event callback, we'll now be able to have our front-end user state maintained even if the user navigates away or closes the page. You'll notice that we're relying on the `user` data in local storage as an indicator that the user is authenticated but that this wouldn't necessarily mean their token is valid. We could write logic within the event callback to make an `$http` request to the server to check the validity of the user's token, but this would create a lot of unnecessary requests. If the user's token is invalid they will receive an error the next time they try to move forward in the application and since these errors are picked up by the `$http.interceptor`, the user will be redirected to the `auth` state on their next move anyway.

Wrapping Up

Hopefully this and the tutorial on Scotch.io were helpful and have given you an idea of how to handle token-based authentication with AngularJS and Laravel. There are a few different ways that authentication can be handled with the two frameworks, but protecting the API with JSON Web Tokens and not relying on session-based authentication gives us the flexibility to be able to use our API for other purposes later on.

Feel free to leave any feedback or questions in the comments below and let me know if there's anything you need help with or if I can clarify anything.

Also, you should follow me on Twitter—I'd love to hear about what you're working on!

TWITTER FACEBOOK GOOGLE+

AngularJS Laravel

Permalink: http://ryanchenkie.com/token-based-authentication-for-angularjs-and-laravel-apps/

Scotch.io | Build an App With Vue.js

Ryan Chenkie

August 31st, 2015

Auth0 | Creating Your First Aurelia App

Audio Processing Your First Adonis App

Ryan Chenkie

August 12th, 2015

NAVIGATION

Copyright © Ryan Chenkie. 2015 • All rights reserved.