Techniques for authentication in AngularJS applications

A collection of ideas for authentication & access control

The Netherlands knows a relatively active AngularJS community, with regular meetups and a helpful Google+ group. Last week I had the opportunity to speak at one of these meetups. Since pretty much all of the applications I build require authentication of some sort, I've implemented various forms of authentication in AngularJS. I decided to share some of these approaches in my talk, and I'm sharing them again in this article.

> *Update: This article is getting very popular. It's been in the Medium top 100 twice now and it's being retweeted and favorited by new people each day. Thanks! The article has also received some comments (the little chatbubbles in the sideline) with several people asking for more explanation on some parts. I noticed the main thing people struggle with is how and where to assign the currentUser object, so I decided to clarify that a bit. Meanwhile, I've switched jobs. I now work at Xebia, a software development and consulting firm with a strong focus on craftsmanship and personal growth. One of our driving values is "Quality without compromise", which I think is essential.*

. . .

Authentication

The most common form of authentication is logging in with a username (or email address) and password. This means implementing a login form where users can enter their credentials. Such a form could look like this:

```
<form name="loginForm" ng-controller="LoginController"
      ng-submit="login(credentials)" novalidate>

  <label for="username">Username:</label>
  <input type="text" id="username"
         ng-model="credentials.username">

  <label for="password">Password:</label>
  <input type="password" id="password"
         ng-model="credentials.password">

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

</form>
```

(Note: there's an extended version below, this is just a simple example)

Since this is an Angular-powered form, we use the ngSubmit directive to trigger a scope function on submit. Note that we're passing the credentials as an argument rather than relying on $scope.credentials, this makes the function easier to unit-test and avoids coupling between the function and it's surrounding scope. The corresponding controller could look like this:

```
.controller('LoginController', function ($scope, $rootScope,
AUTH_EVENTS, AuthService) {
  $scope.credentials = {
    username: '',
    password: ''
  };
  $scope.login = function (credentials) {
    AuthService.login(credentials).then(function (user) {
      $rootScope.$broadcast(AUTH_EVENTS.loginSuccess);
      $scope.setCurrentUser(user);
    }, function () {
      $rootScope.$broadcast(AUTH_EVENTS.loginFailed);
    });
  };
})
```

The first thing to notice is the absence of any real logic. This was done deliberately so to decouple the form from the actual authentication logic. It's

usually a good idea to abstract away as much logic as possible from your controllers, by putting that stuff in services. AngularJS controllers should only manage the $scope object (by watching and manipulating) and not do any heavy lifting.

Communicating session changes

Authenticating is one of those things that affect the state of the entire application. For this reason I prefer to use events (with $broadcast) to communicate changes in the user session. It's a good practice to define all of the available event codes in a central place. I like to use constants for these things:

```
.constant('AUTH_EVENTS', {
  loginSuccess: 'auth-login-success',
  loginFailed: 'auth-login-failed',
  logoutSuccess: 'auth-logout-success',
  sessionTimeout: 'auth-session-timeout',
  notAuthenticated: 'auth-not-authenticated',
  notAuthorized: 'auth-not-authorized'
})
```

A nice thing about constants is that they can be injected like services, which makes them easy to mock in your unit tests. It also allows you to easily rename them (change the values) later without having to change a bunch of files. The same trick is used for user roles:

```
.constant('USER_ROLES', {
  all: '*',
  admin: 'admin',
  editor: 'editor',
  guest: 'guest'
})
```

If you ever want to give all editors the same rights as administrators, you can simply change the value of *editor* to 'admin'.

The AuthService

The logic related to authentication and authorization (access control) is best grouped together in a service:

```
.factory('AuthService', function ($http, Session) {
  var authService = {};

  authService.login = function (credentials) {
    return $http
      .post('/login', credentials)
      .then(function (res) {
        Session.create(res.data.id, res.data.user.id,
                       res.data.user.role);
        return res.data.user;
      });
  };

  authService.isAuthenticated = function () {
    return !!Session.userId;
  };

  authService.isAuthorized = function (authorizedRoles) {
    if (!angular.isArray(authorizedRoles)) {
      authorizedRoles = [authorizedRoles];
    }
    return (authService.isAuthenticated() &&
      authorizedRoles.indexOf(Session.userRole) !== -1);
  };

  return authService;
})
```

To further separate concerns regarding authentication, I like to use another service (a singleton object, using the service style) to keep the user's session information. The specifics of this object depends on your back-end implementation, but I've included a generic example below.

```
.service('Session', function () {
  this.create = function (sessionId, userId, userRole) {
    this.id = sessionId;
    this.userId = userId;
    this.userRole = userRole;
  };
  this.destroy = function () {
    this.id = null;
    this.userId = null;
    this.userRole = null;
```

```
  };
})
```

Once a user is logged in, his information should probably be displayed somewhere (e.g. in the top-right corner). In order to do this, the user object must be referenced in the $scope object, preferably in a place that's accessible to the entire application. While $rootScope would be an obvious first choice, I try to refrain from using $rootScope too much (actually I use it only for global event broadcasting). Instead my preference is to define a controller on the root node of the application, or at least somewhere high up in the DOM tree. The body tag is a good candidate:

```
<body ng-controller="ApplicationController">
  ...
</body>
```

The ApplicationController is a container for a lot of global application logic, and an alternative to Angular's *run* function. Since it's at the root of the $scope tree, all other scopes will inherit from it (except isolate scopes). It's a good place to define the currentUser object:

```
.controller('ApplicationController', function ($scope,
                                               USER_ROLES,
                                               AuthService) {
  $scope.currentUser = null;
  $scope.userRoles = USER_ROLES;
  $scope.isAuthorized = AuthService.isAuthorized;

  $scope.setCurrentUser = function (user) {
    $scope.currentUser = user;
  };
})
```

We're not actually *assigning* the currentUser object, we're merely *initializing* the property on the scope so the currentUser can later be accessed throughout the application. Unfortunately we can't simply assign a new value to it from a child scope, because that would result in a shadow

property. It's a consequence of primitive types (strings, numbers, booleans, undefined and null) being passed by value instead of by reference. To circumvent shadowing, we have to use a setter function. For a lot more on Angular scope and prototypal inheritance, read Understanding Scopes.

Besides initializing the currentUser property, I've also included some properties which allow easy access to USER_ROLES and the isAuthorized function. These should only be used in template expressions, not from other controllers, because doing so would complicate the controller's testability. In the next chapter you'll see how we use these properties.

. . .

Access control

Authorization a.k.a. access control in AngularJS doesn't really exist. Since we're talking about a client-side application, all of the source code is in the client's hands. There's nothing preventing the user from tampering with that code to gain 'access' to certain views and interface elements. All we can really do is **visibility control**. If you need real authorization you'll have to do it server-side, but that's beyond the scope of this article.

Restricting element visibility

AngularJS comes with several directives to show or hide an element based on some scope property or expression: ngShow, ngHide, ngIf and ngSwitch. The first two will use a *style* attribute to hide the element, while the last two will actually remove the element from the DOM.

The first solution (hiding it) is best used only when the expression changes frequently and the element doesn't contain a lot of template logic and scope references. The reason for this is that any template logic within a hidden element will still be reevaluated on each digest cycle, slowing down the application. The second solution will remove the DOM element entirely, including any event handlers and scope bindings. Changing the DOM is a lot of work for the browser (hence the reason for using ngShow/ngHide in some cases), but worth the effort most of the time. Since user access doesn't change often, using ngIf or ngSwitch is the best choice:

```
<div ng-if="currentUser">Welcome, {{ currentUser.name }}</div>
```

```
<div ng-if="isAuthorized(userRoles.admin)">You're admin.</div>
```

```
<div ng-switch on="currentUser.role">
  <div ng-switch-when="userRoles.admin">You're admin.</div>
  <div ng-switch-when="userRoles.editor">You're editor.</div>
  <div ng-switch-default>You're something else.</div>
</div>
```

The switch example assumes a user can have only one role. I'm sure you can come up with something more flexible, but you get the idea.

Restricting route access

Most of the time you will want to disallow access to an entire page rather than hide a single element. Using a custom data object on the route (or *state*, when using UI Router), we can specify which roles should be allowed access. This example uses the UI Router style, but the same will work for ngRoute.

```
.config(function ($stateProvider, USER_ROLES) {
  $stateProvider.state('dashboard', {
    url: '/dashboard',
    templateUrl: 'dashboard/index.html',
    data: {
      authorizedRoles: [USER_ROLES.admin, USER_ROLES.editor]
    }
  });
})
```

Next, we need to check this property every time the route changes (i.e. the user navigates to another page). This involves listening to the $routeChangeStart (for ngRoute) or $stateChangeStart (for UI Router) event:

```
.run(function ($rootScope, AUTH_EVENTS, AuthService) {
  $rootScope.$on('$stateChangeStart', function (event, next) {
    var authorizedRoles = next.data.authorizedRoles;
```

```
      if (!AuthService.isAuthorized(authorizedRoles)) {
        event.preventDefault();
        if (AuthService.isAuthenticated()) {
          // user is not allowed
          $rootScope.$broadcast(AUTH_EVENTS.notAuthorized);
        } else {
          // user is not logged in
          $rootScope.$broadcast(AUTH_EVENTS.notAuthenticated);
        }
      }
    });
  })
```

When a user is not authorized to access the page (because he's not logged in or doesn't have the right role), the transition to the next page will be prevented, so the user will stay on the current page. Next, we broadcast an event which other modules can listen to. I suggest including a loginDialog directive on the page which appears when the notAuthenticated event is fired, and an error message which should appear when the notAuthorized event occurs.

. . .

Session expiration

Authentication is mostly a server-side affair. Whatever way you implement it, your back-end will have to do the actual verification of user credentials and handle things like session expiration and revoking access. This means your API will sometimes respond with an authentication error. The standard way to communicate these is by using HTTP status codes. Commonly used status codes for authentication errors are:

- 401 Unauthorized—The user is not logged in
- 403 Forbidden—The user is logged in but isn't allowed access
- 419 Authentication Timeout (non standard)—Session has expired
- 440 Login Timeout (Microsoft only)—Session has expired

The last two are not part of any standard, but may be used in the wild. The best official way to communicate a session timeout would be a 401. Either way, your loginDialog should automatically appear as soon as the API

returns a 401, 419 or 440 and an error should be shown on a 403. Basically we want to broadcast the same notAuthenticated / notAuthorized event based on the HTTP response status code. For this we can add an interceptor to $httpProvider:

```
.config(function ($httpProvider) {
  $httpProvider.interceptors.push([
    '$injector',
    function ($injector) {
      return $injector.get('AuthInterceptor');
    }
  ]);
})
```

```
.factory('AuthInterceptor', function ($rootScope, $q,
                                      AUTH_EVENTS) {
  return {
    responseError: function (response) {
      $rootScope.$broadcast({
        401: AUTH_EVENTS.notAuthenticated,
        403: AUTH_EVENTS.notAuthorized,
        419: AUTH_EVENTS.sessionTimeout,
        440: AUTH_EVENTS.sessionTimeout
      }[response.status], response);
      return $q.reject(response);
    }
  };
})
```

This is just a simple implementation of an AuthInterceptor. There's a great project on Github that does the same thing but also includes an httpBuffer service which—when an HTTP error occurs—will prevent further API requests until the user is logged in again, then re-fire them in order.

The loginDialog directive

When a session expires, we need the user to reenter his credentials. To prevent him from losing his work, it's best to show the login form in a dialog box rather than redirect to the login page. This dialog box should listen to the notAuthenticated and sessionTimeout events, so it opens itself when one of these events is triggered.

```
.directive('loginDialog', function (AUTH_EVENTS) {
  return {
    restrict: 'A',
    template: '<div ng-if="visible"
                    ng-include="\'login-form.html\'">',
    link: function (scope) {
      var showDialog = function () {
        scope.visible = true;
      };

      scope.visible = false;
      scope.$on(AUTH_EVENTS.notAuthenticated, showDialog);
      scope.$on(AUTH_EVENTS.sessionTimeout, showDialog)
    }
  };
})
```

This of course can be extended in any way you like. The main idea is to reuse the existing login form template and it's LoginController. You'll need something like this on each page (e.g. right before </body>):

```
<div login-dialog ng-if="!isLoginPage"></div>
```

Note the *isLoginPage* check. A failed login will trigger a notAuthenticated event, but we don't want to show the dialog on the login page because that would be redundant and odd. That's why we won't include the loginDialog on the login page. It makes sense to define $scope.isLoginPage in the ApplicationController.

. . .

Issues with the login form

Many users will rely on a password manager to store their credentials so they can easily login again later. If you were to use the simple example at the start of this article, you'd find that password managers have a hard time dealing with AngularJS forms. There's two problems that may occur:

- Form submission is not detected so credentials are never stored

- Automatically filled fields are not picked up by Angular

These problems can be circumvented, but that involves an iframe and a timeout function. If you prefer to keep your code clean from ugly hacks, you may want to keep the login form outside of AngularJS completely and rely on old-fashioned server-side rendering instead. Otherwise, here's the 'improved' form:

```
<iframe src="sink.html" name="sink"
        style="display:none"></iframe>

<form name="loginForm"
      action="sink.html" target="sink" method="post"
      ng-controller="LoginController"
      ng-submit="login(credentials)"
      novalidate form-autofill-fix>

  <label for="username">Username:</label>
  <input type="text" id="username"
         ng-model="credentials.username">

  <label for="password">Password:</label>
  <input type="password" id="password"
         ng-model="credentials.password">

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

</form>
```

The difference is in the newly introduced <iframe> and the added attributes on the <form> element. By providing an action, target and method, the form will be posted to sink.html in the iframe, aside from calling the ngSubmit function. This way the browser's normal form processing logic will kick in, without navigating to sink.html (at least not in the main window). Password managers will recognize this as a regular form submission, and ask to store the credentials on the keychain. The sink.html page is just an empty HTML document, sitting beside your index.html. I like to have a comment in there explaining why the file exists (so another developer won't remove it).

> *Don't forget to include method="post", otherwise your credentials will be appended as querystring parameters, which are exposed in the browser's history.*

Now that the password manager is able to store the credentials, we have to support restoring these credentials. This is called autofill (as opposed to autocomplete). The problem with autofill is that most browsers don't trigger an event on the input field they have autofilled. Because of that, AngularJS has no way of knowing that the field's contents have changed, and thus will not update the $scope object with the new value. The result is submitting a supposedly complete login form, only to be denied access for having supplied invalid credentials. Here's where the formAutofillFix directive comes in:

```
.directive('formAutofillFix', function ($timeout) {
  return function (scope, element, attrs) {
    element.prop('method', 'post');
    if (attrs.ngSubmit) {
      $timeout(function () {
        element
          .unbind('submit')
          .bind('submit', function (event) {
            event.preventDefault();
            element
              .find('input, textarea, select')
              .trigger('input')
              .trigger('change')
              .trigger('keydown');
            scope.$apply(attrs.ngSubmit);
          });
      });
    }
  };
});
```

This directive will rebind the submit event with a function that triggers an event on each of the input fields (forcing Angular to update the scope), waits for Angular to finish the digest cycle, then calls the ngSubmit function. Any scope properties are updated with the autofilled values, the moment the form is submitted. A drawback of this approach is that form validation will not work, because the autofilled values are not available on the scope before the form is submitted. If you need that to work, there's a polyfill which triggers change events earlier on.

. . .

Restoring user state

A tricky part of authentication in a single page application (SPA) is retaining or restoring the information of the logged in user when the visitor refreshes the page. Since all of the state is client-side, refreshing will clear user info. In order to fix this I usually implement an API endpoint (e.g. /profile) which returns the user data when logged in. This endpoint is called when starting the AngularJS application (e.g. in the “run” function). The user data is then saved in a Session service or in $rootScope, just as you would after a successful login. Alternatively you can embed the user data in index.html, so you don’t have to do an additional request. A third solution would be to store the user data in a cookie or Local Storage, but that makes it harder to force a logout and/or forget user data.

When trying to directly (by URL) access a route which requires a user to be logged in, it may happen that the user data is not loaded before the page is rendered. This can be avoided by defining a ‘resolve’ property on the route:

```
$stateProvider.state('protected-route', {
  url: '/protected',
  resolve: {
    auth: function resolveAuthentication(AuthResolver) {
      return AuthResolver.resolve();
    }
  }
});
```

The AuthResolver is a service which watches the value of ‘currentUser’ on $rootScope, and will only resolve after currentUser has been set:

```
.factory('AuthResolver', function ($q, $rootScope, $state) {
  return {
    resolve: function () {
      var deferred = $q.defer();
      var unwatch = $rootScope.$watch('currentUser', function
(currentUser) {
        if (angular.isDefined(currentUser)) {
          if (currentUser) {
            deferred.resolve(currentUser);
```

```
            } else {
              deferred.reject();
              $state.go('user-login');
            }
            unwatch();
          }
        });
        return deferred.promise;
      }
    };
  })
```

Since this watcher will only be used once, it should be disabled afterwards. This can be done by *unwatching*: $watch returns a function which you can call to remove the watcher. You could extend the example above with additional functionality such as a redirect to the original page after login.

. . .

There's countless variations and other options that you can explore. In the end it will mostly depend on the specifics of your back-end implementation. Keep in mind that the only real security is on the server-side. And remember to always use HTTPS.

Follow me on Twitter

Read my other articles:

- Advanced routing and resolves
- Scalable code organization in AngularJS