Cookies vs. Tokens: The Definitive Guide

by Adnan Kukic MVB · Jun. 02, 16 · Integration Zone · Opinion

We will be writing an Angular 2 app that uses JWT for authentication. Grab the Github repo if you would like to follow along.

Our last article comparing cookie to token authentication was over two years ago. Since then, we've written extensively on how to integrate token authentication across many different languages and frameworks.

The rise of single page applications (SPAs) and decoupling of the front-end from the back-end is in full force. Frameworks like Angular, React, and Vue allow developers to build bigger, better, and more performant single page applications than ever before. Token-based authentication goes hand in hand with these frameworks.

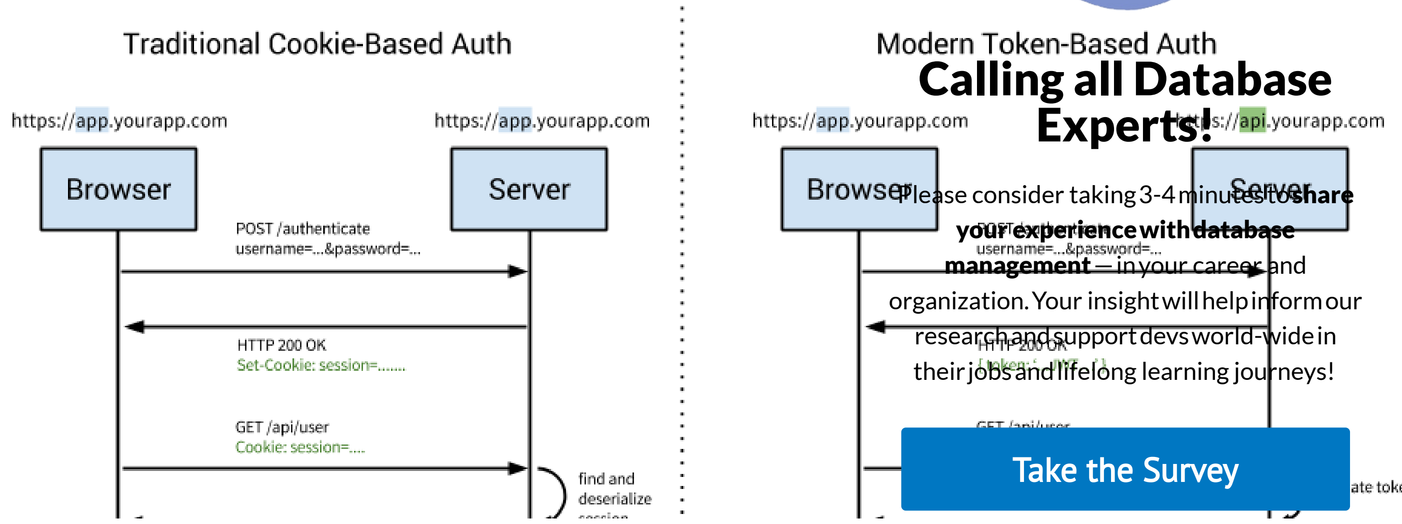
"Token-based authentication goes hand in hand with SPA frameworks like Angular, React and Vue."

TWEET THIS

Cookie vs. Token Authentication - Recap

Before we dive further, let's quickly recap how these two authentication systems work. If you are already familiar with how cookie and token authentication works, feel free to skip this section, otherwise read on for an in-depth overview.

This diagram is a great introduction and simplified overview of the difference between cookie and token approaches to authentication.



Cookie-Based Authentication

Cookie-based authentication has been the default, tried-and-true method for handling user authentication for a long time.

Cookie-based authentication is **stateful**. This means that an authentication record or session must be kept both server and client-side. The server needs to keep track of active sessions in a database, while on the front-end a cookie is created that holds a session identifier, thus the name cookie based authentication. Let's look at the flow of traditional cookie-based authentication:

1. User enters their login credentials.
2. Server verifies the credentials are correct and creates a session which is then stored in a database.
3. A cookie with the session ID is placed in the users browser.
4. On subsequent requests, the session ID is verified against the database and if valid the request processed.
5. Once a user logs out of the app, the session is destroyed both client-side and server-side.

Token-Based Authentication

Token-based authentication has gained prevalence over the last few years due to the rise of single page applications, web APIs, and the Internet of Things (IoT). When we talk about authentication with tokens, we generally talk about authentication with JSON Web Tokens (JWTs). While there are different ways to implement tokens, JWTs have become the de-facto standard. With this context in mind, the rest of the article will use tokens and JWTs interchangeably.

Token-based authentication is **stateless**. The server does not keep a record of which users are logged in or which JWTs have been issued. Instead, every request to the server is accompanied by a token which the s uses to verify the authenticity of the request. The token is generally sent as an addition Authorization head of `Bearer {JWT}`, but can additionally be sent in the body of a POST request or even as a Let's see how this flow works:

1. User enters their login credentials.
2. Server verifies the credentials are correct and returns a signed token.
3. This token is stored client-side, most commonly in local storage - but can be stored in session storage or a cookie as well.
4. Subsequent requests to the server include this token as an additional Authorization header or through one of the other methods mentioned above.
5. The server decodes the JWT and if the token is valid processes the request.
6. Once a user logs out, the token is destroyed client-side, no interaction with the server is necessary.

Advantages of Token-Based Authentication

Understanding **how** something works is only half the battle. Next, we'll cover n is preferable over the traditional cookie-based approach.

Stateless, Scalable, and Decoupled

Perhaps the biggest advantage to using tokens over cookies is the fact that token authentication is stateless. The back-end does not need to keep a record of tokens. Each token is self-contained, containing all the data required to check it's validity as well as convey user information through claims.

The server's only job, then, becomes to sign tokens on a successful login request and verify that incoming tokens are valid. In fact, the server does not even need to sign tokens. Third party services such as Auth0 can handle the issuing of tokens and then the server only needs to verify the validity of the token.

Cross Domain and CORS

Cookies work well with singular domains and sub-domains, but when it comes to managing cookies across different domains, it can get hairy. In contrast, a token-based approach with CORS enabled makes it trivial to expose APIs to different services and domains. Since the JWT is required and checked with each and every call to the back-end, as long as there is a valid token, requests can be processed. There are a few caveats to this and we'll address those in the Common Questions and Concerns section below.

Store Data in the JWT

With a cookie based approach, you simply store the session id in a cookie. JWT's, on the other hand, allow you to store any type of metadata, as long as it's valid JSON. The JWT spec specifies different types of claims that can be included such as reserved, public and private. You can learn more about the specifics and the differences between the types of claims on the jwt.io website.

In practice, what this means is that a JWT can contain any type of data. Depending on your use case you may choose to make the minimal amount of claims such as the user id and expiration of the token, or you may decide to include additional claims such as the user's email address, who issued the token, scopes or permissions for the user, and more.

Performance

When using the cookie-based authentication, the back-end has to do a lookup, whether that be a traditional SQL database or a NoSQL alternative, and the round trip is likely to take longer compared to decoding a token. Additionally, since you can store additional data inside the JWT, such as the user's permission level, yo[illegible]urself additional lookup calls to get and process the requested data.

For example, say you had an API resource `/api/orders` that retrieves the latest orders [illegible] app, but only users with the role of **admin** have access to view this data. In a cookie based approach, one [illegible]est is made, you'd have one call to the database to verify that the session is valid, another to get the user data and verify that the user has the role of **admin**, and finally a third call to get the data. On the other hand, with a JWT approach, you can store the user role in the JWT, so once the request is made and the JWT verified, you can make a single call to the database to retrieve the orders.

Mobile Ready

Modern APIs do not only interact with the browser. Written properly a single API can serve both the browser and native mobile platforms like iOS and Android. Native mobile platforms and cookies do not mix well. While possible, there are many limitations and considerations to using cookies with mobile platforms. Tokens, on the other hand, are much easier to implement on both iOS and Android. Tokens are also easier to implement for Internet of Things applications and services that do not have a concept of a cookie store.

Common Questions and Concerns

Common Questions and Concerns

In this section, we'll take a look at some common questions and concerns that frequently arise when the topic of token authentication comes up. The key focus here will be security but we'll examine use cases concerning token size, storage and encryption.

JWT Size

The biggest disadvantage of token authentication is the size of JWTs. A session cookie is relatively tiny compared to even the smallest JWT. Depending on your use case, the size of the token could become problematic if you add many claims to it. Remember, each request to the server must include the JWT along with it.

Where to Store Tokens?

With token-based auth, you are given the choice of where to store the JWT. Commonly, the JWT is placed in the browser's local storage and this works well for most use cases. There are some issues with storing JWTs in local storage to be aware of. Unlike cookies, local storage is sandboxed to a specific domain and its data cannot be accessed by any other domain including sub-domains.

You can store the token in a cookie instead, but the max size of a cookie is only 4kb so that may be problematic if you have many claims attached to the token. Additionally, you can store the token in session storage which is similar to local storage but is cleared as soon as the user closes the browser.

XSS and XSRF Protection

Protecting your users and servers is always a top priority. One of the most common concerns developers have when deciding on whether to use token-based authentication is the security implications. Two of the most common attack vectors facing websites are Cross Site Scripting (XSS) and Cross-Site Request Forgery (XSRF or CSRF).

Cross Site Scripting) attacks occur when an outside entity is able to execute code within your website or app. The most common attack vector here is if your website allows inputs that are not properly sanitized. If an attacker can execute code on your domain, your JWT tokens are vulnerable. Our CTO has argued in the past that XSS attacks are much easier to deal with compared to XSRF attacks because they are generally better understood. Many frameworks, including Angular, automatically sanitize inputs and prevent arbitrary code execution. If you are not using a framework that sanitizes input/output out-of-the-box, you can look at plugins like caja developed by Google to assist. Sanitizing inputs is a solved issue in many frameworks and languages and I would recommend using a fra plugin vs building your own.

Cross Site Request Forgery attacks are not an issue if you are using JWT with local stora hand, if your use case requires you to store the JWT in a cookie, you will need to protect against XSRF. X as easily understood as XSS attacks. Explaining how XSRF attacks work can be time-consuming, so instead, check out this really good guide that explains in-depth how XSRF attacks work. Luckily, preventing XSRF attacks is a fairly simple matter. To over-simplify, protecting against an XSRF attack, your server, upon establishing a session with a client will generate a unique token (note this is not a JWT). Then, anytime data is submitted to your server, a hidden input field will contain this token and the server will check to make sure the tokens match. Again, as our recommendation is to store the JWT in local storage, you probably will not have to worry about XSRF attacks.

One of the best ways to protect your users and servers is to have a short expiration time for tokens. That way, even if a token is compromised, it will quickly become useless. Additionally, you may maintain a blacklist of compromised tokens and not allow those tokens access to the system. Finally, the nuclear approach would be to change the signing algorithm, which would invalidate all active tokens and require all of your users to log in again. This approach is not easily recommended, but is available in the event of a severe breach.

Tokens Are Signed, Not Encrypted

A JSON Web Token is comprised of three parts: the header, payload, and signature. The format of a JWT is `header.payload.signature`. If we were to sign a JWT with the HMACSHA256 algorithm, the secret 'shhhh' and the payload of:

```
{
  "sub": "1234567890",
  "name": "Ado Kukic",
  "admin": true
}
```

The JWT generated would be:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkFkbyBLdWtpYyIsImFkbWluIjp0cnVlLCJpYXQiOjE0NjQyOTc4ODV9.Y4
```

The **very** important thing to note here is that this token is signed by the HMACSHA256 algorithm, and the header and payload are Base64URL encoded, it is **not** encrypted. If I go to jwt.io, paste this token and select the HMACSHA256 algorithm, I could decode the token and read its contents. Therefore, it should go without saying that sensitive data, such as passwords, should never be stored in the payload.

If you must store sensitive data in the payload or your use case calls for the JWT to be obscured, you can use JSON Web Encryption (JWE). JWE allows you to encrypt the contents of a JWT so that it is not readable by anyone but the server. JOSE provides a great framework and different options for JWE and has SDKs for many popular frameworks including NodeJS and Java.

Token-Based Authentication in Action with Auth0

Here at Auth0, we've written SDKs, guides, and quickstarts for working with JWTs for many languages and frameworks including NodeJS, Java, Python, GoLang, and many more. Our last "Cookies vs. Tokens" article used the AngularJS framework, so it's fitting to use Angular 2 for our code samples today.

You can download the sample code from our GitHub repo. Downloading the code samples is preferable as Angular 2 requires a lot of initial setup to get going. If you haven't already, sign up for a free Auth0 account so you can do the implementation yourself and experiment with different features and options. Let's get sta

Setting Up the Back-end Server

We'll first set up our server. We'll build our server with NodeJS. The server will expose g and `/secured/ping`. The first will be publicly available any anyone can access it, while the sec es you to be authenticated to call it. The implementation is below:

```
// Include the modules needed for our server.
var http = require('http');
var express = require('express');
var cors = require('cors');
var app = express();
var jwt = require('express-jwt');

// Set up our JWT authentication middleware
// Be sure to replace the YOUR_AUTH0_CLIENT_SECRET and
// YOUR_AUTHO_CLIENT_ID with your apps credentials which
// can be found in your management dashboard at
// https://manage.auth0.com
var authenticate = jwt({
  secret: new Buffer('YOUR_AUTH0_CLIENT_SECRET', 'base64'),
  audience: 'YOUR_AUTH0_CLIENT_ID'
});
```

```

app.use(cors());

// Here we have a public route that anyone can access
app.get('/ping', function(req, res) {
  res.send(200, {text: "All good. You don't need to be authenticated to call this"});
});

// We include the authenticate middleware here that will check for
// a JWT and validate it. If there is a token and it is valid the
// rest of the code will execute.
app.get('/secured/ping', authenticate, function(req, res) {
  res.send(200, {text: "All good. You only get this message if you're authenticated"});
})

var port = process.env.PORT || 3001;

// We launch our server on port 3001.
http.createServer(app).listen(port, function (err) {
  console.log('listening in http://localhost:' + port);
});
```

This is a pretty standard Node/Express setup. The only unique thing we did was implement the `express-jwt` middleware which will validate a JWT. Since we are doing this integration with Auth0, we'll let Auth0 handle the process of generating and signing tokens. If we did not want to use Auth0, we could create and sign our own tokens with the `jsonwebtoken` module. Let's see an example of how this can be accomplished.

```
// Import modules
...
var jwt = require('jsonwebtoken');

var token = jwt.sign({ sub : "1234567890", name : "Ado Kukic", admin: true }, 'shhhh');

app.get('/token', function(req, res){
    res.send(token);
});
```

If we were to write this code, launch the server, and navigate to `localhost:3001/token` we would see a signed token containing the three claims we made. The `jsonwebtoken` module can also be used to verify and decrypt the tokens. To learn more about it, check out its repo. As we won't be generating tokens on our server, we ove this code.

Implementing the Front-end

Next, we'll implement our Angular 2 app. If you are following along from our GitHub re wo options for the front-end. One uses systemjs while the other Webpack for loading and managing our s the preferred way to write Angular 2 apps is with TypeScript, we'll build our sample app with TypeScript. For our demo, we'll be working out of the **systemjs** directory. Additionally, we'll be using the `angular2-jwt` library which provides helper methods for making requests with the correct headers and also checking to see if a valid token exists.

First things first. We need an entry point into our app and that is `index.html`.

```
<html>
  <head>
    <base href="/">
    <title>Angular 2 Playground</title>
      <meta charset="UTF-8">
     <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">

     <!-- We'll include the Auth0 Lock widget to handle authentication -->
     <script src="//cdn.auth0.com/js/lock-9.0.min.js"></script>

    <script src="node_modules/es6-shim/es6-shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/reflect-metadata/Reflect.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>

    <script src="systemjs.config.js"></script>
    <script>
      System.import('app').catch(function(err){ console.error(err); });
    </script>
  </head>
  <body>
    <app>Loading...</app>
  </body>
</html>
```

Next, we'll define the entry point of our Angular 2 application. This will be done in a file called `main.ts`.

```
import { bootstrap } from '@angular/platform-browser-dynamic';
import {provide} from '@angular/core';
import {LocationStrategy, HashLocationStrategy} from '@angular/common';
import {RouteConfig, ROUTER_PROVIDERS, ROUTER_DIRECTIVES} from '@angular/router-deprecated';
import {HTTP_PROVIDERS} from '@angular/http';
// Here we load the angular2-jwt library
import {AUTH_PROVIDERS} from 'angular2-jwt';

import { App } from './app.component';

bootstrap(App, [
  HTTP_PROVIDERS,
  ROUTER_PROVIDERS,
  AUTH_PROVIDERS,
  provide(LocationStrategy, { useClass: HashLocationStrategy })
])
```

We'll build the `auth.service.ts` next. This service will provide methods for logging users in and out of our application. Be sure to replace the `YOUR_CLIENT_ID` and `YOUR_DOMAIN` with your apps settings from your Auth0 management dashboard.

```
import {Injectable, NgZone} from '@angular/core';
import {Router} from '@angular/router-deprecated';
import {AuthHttp, tokenNotExpired} from 'angular2-jwt';

// Avoid name not found warnings
declare var Auth0Lock: any;

@Injectable()
export class Auth {
  // Replace YOUR_CLIENT_ID and YOUR_DOMAIN with your credentials
  lock = new Auth0Lock('YOUR_CLIENT_ID', 'YOUR_DOMAIN');
  refreshSubscription: any;
  user: Object;
  zoneImpl: NgZone;

  constructor(private authHttp: AuthHttp, zone: NgZone, private router: Router) {
    this.zoneImpl = zone;
    this.user = JSON.parse(localStorage.getItem('profile'));
  }

  public authenticated() {
    // Check if there's an unexpired JWT
    return tokenNotExpired();
  }

  public login() {
    // Show the Auth0 Lock widget
    this.lock.show({}, (err, profile, token) => {
      if (err) {
        alert(err);
        return;
      }
      // If authentication is successful, save the items
      // in local storage
      localStorage.setItem('profile', JSON.stringify(profile));
      localStorage.setItem('id_token', token);
      this.zoneImpl.run(() => this.user = profile);
    });
  }

  public logout() {
    localStorage.removeItem('profile');
    localStorage.removeItem('id_token');
    this.zoneImpl.run(() => this.user = null);
    this.router.navigate(['Home']);
  }
}
```

Now that we have the foundation complete. We can start building our application. We'll build our root component in a file called `app.component.ts` .

```
import {Component} from '@angular/core';
import {RouteConfig, ROUTER_PROVIDERS, ROUTER_DIRECTIVES} from '@angular/router-deprecated';
import {HTTP_PROVIDERS} from '@angular/http';
import {AUTH_PROVIDERS} from 'angular2-jwt';

import {Home} from './home.component';
import {Ping} from './ping.component';
import {Profile} from './profile.component';
import {Auth} from './auth.service';

@Component({
  selector: 'app',
  providers: [ Auth ],
  directives: [ ROUTER_DIRECTIVES ],
  templateUrl: 'src/app.template.html',
  styles: [`.btn-margin { margin-top: 5px}`]
})
@RouteConfig([
  { path: '/home',  name: 'Home',  component: Home, useAsDefault: true },
  { path: '/ping',  name: 'Ping',  component: Ping },
  { path: '/profile',  name: 'Profile',  component: Profile }
])
export class App {

  constructor(private auth: Auth) {}

}
```

You may notice from our directive metadata that we will be loading an external template called `app.template.html` . Angular 2 templates can be inlined or reference an external file and since our template is on the longer side, we'll place it in an external file.

```
<nav class="navbar navbar-default">
  <div class="container-fluid">
    <div class="navbar-header">
      <a class="navbar-brand" href="#">Auth0 - Angular 2</a>
      <button class="btn btn-primary btn-margin" [routerLink]=" ['Home'] ">Home</button>
      <button class="btn btn-primary btn-margin" [routerLink]=" ['Ping'] ">Ping</button>
      <button class="btn btn-primary btn-margin" [routerLink]=" ['Profile'] " *ngIf="auth.authenticated()">Profile</button>
      <button class="btn btn-primary btn-margin" (click)="auth.login()" *ngIf="!auth.authenticated()">Log In</button>
      <button class="btn btn-primary btn-margin" (click)="auth.logout()" *ngIf="auth
    </div>
  </div>
</nav>
```

```
</nav>

<main class="container">
  <router-outlet></router-outlet>
</main>
```

From our `app.template.html` file, we see that we will end up having three routes: **Home**, **Ping,** and **Profile**. Additionally, there are two more buttons, Log In and Log Out. The `*ngIf` directive conditionally displays some routes based on whether the user is authenticated or not. Let's build out the three routes.

Home Component

The home component is the default route loaded. It is publicly accessible.

```
import {Component} from '@angular/core';

@Component({
  selector: 'home',
  template: `
    <h1>Welcome to auth0-angular2</h1>
    <p>
      This repo shows you how to set up authentication in your Angular 2 apps with Auth0.
      Get started by providing your Auth0 client ID and domain in the Auth0Lock widget in <code>auth/auth.service.ts</code>.
    </p>
  `
})
export class Home {
  constructor() {}
}
```

Ping Component

The ping component interacts with our NodeJS server that we built earlier. The Node server will need to be running for you to access these routes.

```
import {Component} from '@angular/core';
import {Http} from '@angular/http';

import {AuthHttp} from 'angular2-jwt';
import {Auth} from './auth.service';
import 'rxjs/add/operator/map';

@Component({
  selector: 'ping',
  template: `
    <h1>Send a Ping to the Server</h1>
    <p *ngIf="!auth.authenticated()">Log In to Get Access to a Secured Ping</p>
    <button class="btn btn-primary" (click)="ping()">Ping</button>
    <button class="btn btn-primary" (click)="securedPing()" *ngIf="auth.authenticated()">Secured Ping</button>
    <h2></h2>
  `
})
export class Ping {
  API_URL: string = 'http://localhost:3001';
  message: string;

  constructor(private http: Http, private authHttp: AuthHttp, private auth: Auth) {}

  ping() {
    this.http.get(`${this.API_URL}/ping`)
      .map(res => res.json())
      .subscribe(
        data => this.message = data.text,
        error => this.message = error._body
      );
  }

  securedPing() {
    this.authHttp.get(`${this.API_URL}/secured/ping`)
      .map(res => res.json())
      .subscribe(
        data => this.message= data.text,
        error => this.message = error._body
      );
  }
}
```

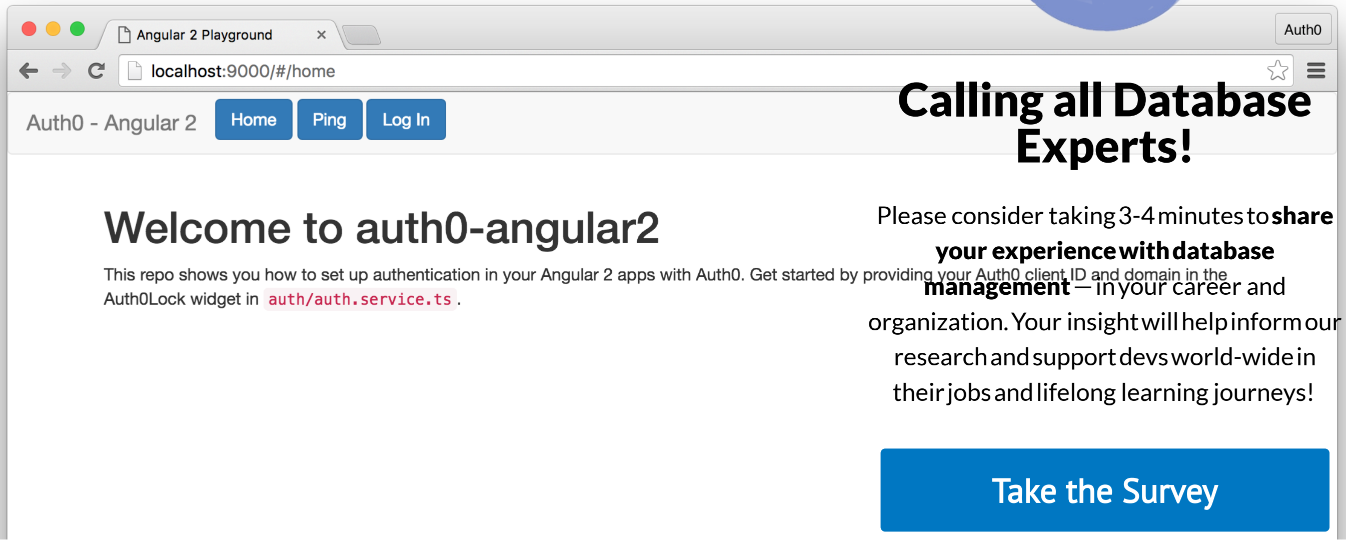
Profile Component

The profile component displays user data for the currently logged in user. This component can only be accessed by a logged in user.

```
import {Component} from '@angular/core';
import {CanActivate} from '@angular/router-deprecated';
import {tokenNotExpired} from 'angular2-jwt';
import {Auth} from './auth.service';

@Component({
  selector: 'profile',
  template: `
    <h1>Profile</h1>
    <img [src]="auth.user.picture">
    <h2></h2>
    <span></span>
  `
})

@CanActivate(() => tokenNotExpired())

export class Profile {
  constructor(private auth: Auth) {}
}
```

With the three components built, we are ready to launch our app. If you are using the provided GitHub repo, you can simply run `gulp play` and your code will be transpiled into JavaScript and the application will launch at `localhost:9000`. If you did not use the GitHub repo, you will need to transpile the typescript or change the systemjs configuration to load typescript files instead.

Navigating to `localhost:9000` will load our the UI for app which will look like:



Welcome to auth0-angular2

This repo shows you how to set up authentication in your Angular 2 apps with Auth0. Get started by providing your Auth0 client ID and domain in the Auth0Lock widget in `auth/auth.service.ts` .

You can navigate to the **Ping** tab and click the "Ping" button to make a call to your Node API and it will display the correct message. If you click on the **Login** button you will be prompted to log in using the Auth0 Lock widget. Upon successfully authenticating, you will be able to view your Profile, log out, and have the ability to call a **Secured Ping** from the Ping tab.

Progressive Web Apps

The last topic I want to cover before we close out this article is Progressive Web Apps. Progressive Web Apps allow your web based application to behave more like a native mobile iOS or Android app. Progressive Web Apps bring many advantages including increased performance, are "installable" on mobile devices, and can work offline.

Angular 2, through its Mobile Toolkit, makes it easy to transform your Angular 2 app into a Progressive Web App. There are many components that can make your app more progressive, the one we'll look at today is the **webapp manifest**. This manifest is simply a file, similar to `package.json` for example, where you define specifics for your application. When your website is accessed on a mobile device, this manifest can be read and based on the information inside certain actions taken like setting the app name or setting the orientation of the app. Let's look at an app manifest, which is titled `manifest.webapp` and see which options we can set:

```
{
  "name": "Auth0 Angular 2 App",
  "short_name": "A0 Angular 2 App",
  "icons": [
    {
            "src": "/android-chrome-36x36.png",
            "sizes": "36x36",
            "type": "image/png",
            "density": 0.75
        },
        {
            "src": "/android-chrome-48x48.png",
            "sizes": "48x48",
            "type": "image/png",
            "density": 1
        },
        {
            "src": "/android-chrome-72x72.png",
            "sizes": "72x72",
            "type": "image/png",
            "density": 1.5
        },
        {
            "src": "/android-chrome-96x96.png",
            "sizes": "96x96",
            "type": "image/png",
            "density": 2
        },
        {
```

```
        {
            "src": "/android-chrome-144x144.png",
            "sizes": "144x144",
            "type": "image/png",
            "density": 3
        },
        {
            "src": "/android-chrome-192x192.png",
            "sizes": "192x192",
            "type": "image/png",
            "density": 4
        }
  ],
  "theme_color": "#000000",
  "background_color": "#e0e0e0",
  "start_url": "/index.html",
  "display": "standalone",
  "orientation": "portrait"
}
```

Conclusion

In today's article we compared the differences between cookie- and token-based authentication. We highlighted the advantages and concerns of using tokens and also wrote a simple app to showcase how JWT works in practice. There are many reasons to use tokens and Auth0 is here to ensure that implementing token authentication is easy and secure. Finally, we introduced Progressive Web Apps to help make your web applications feel more native on mobile devices. Sign up for a free account today and be up and running in minutes.

Like This Article? Read More From DZone

DZone Article
Stop Using JWTs as Session Tokens

DZone Article
All You Need to Know About User Session Security

DZone Article
OAuth 2.0 vs Session Management

Free DZone Refcard
Introduction to Digital Asset Management via APIs

Topics: APPLICATIONS , AUTHENTICATION , COOKIE , JWT , LOCAL STORAGE , TOKEN , TOKENS

Published at DZone with permission of Adnan Kukic , DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.

SURVEY