medium.com /swlh/everything-you-need-to-know-about-the-passport-jwt-passport-js-strategy-8b69f39014b0

Everything you need to know about the `passport-jwt` Passport JS Strategy (and Angular implementation)

Zach Gollwitzer ∶ 22-28 minutes ∶ 1/14/2020

Passport JWT
Strategy
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxw
RJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Table of Contents

What is JWT Based Authentication?

Before I start getting lost in the details, I must mention that if you read my post on the `passport-local` authentication strategy, this post will be much much easier! We have already covered a lot of the topics needed to understand how to use the `passport-jwt` authentication strategy in that post.

Additionally, as we walk through the basics of JWT authentication, we will start to understand why JWT authentication is far better for Angular front-end applications than session-based authentication (hint: stateless auth!).

Review and Preview

As we transition from talking about session-based authentication to JWT based authentication, it is important to keep our authentication flows clear. To do a quick review, the basic auth flow of a session-based authentication app is like so:

1. User visits your Express application and signs in using his username and password
2. The username and password are sent via POST request to the `/login` route on the Express application server
3. The Express application server will retrieve the user from the database (a hash and salt are stored on the user profile), take a hash of the password that the user provided a few seconds ago using the salt attached to the database user object, and verify that the hash taken matches the hash stored on the database user object.
4. If the hashes match, we conclude that the user provided the correct credentials, and our `passport-local` middleware will attach the user to the current session.
5. For every new request that the user makes on the front-end, their session Cookie will be attached to the request, which will be subsequently verified by the Passport middleware. If the Passport middleware verifies the session cookie successfully, the server will return the requested route data, and our authentication flow is complete.

What I want you to notice about this flow is the fact that the user only had to type in his username and password **one time**, and for the remainder of the session, he can visit protected routes. The session cookie is **automatically** attached to all of his requests because this is the default behavior of a web browser and how cookies work! In addition, each time a request is made, the Passport middleware and Express Session middleware will be making a query to our database to retrieve session information. In other words, **to authenticate a user, a database is required**.

Now skipping forward, you'll begin to notice that with JWTs, there is absolutely no database required on **each request** to authenticate users. Yes, we will need to make one database request to initially authenticate a user and generate a JWT, but after that, the JWT will be attached in the `Authorization` HTTP header (as opposed to `Cookie` header), and no database is required.

If this doesn't make sense, that is okay. We will cover all of the logic in the remaining sections.

What is a JWT (JSON Web Token)?

To accommodate those who might already have an understanding of JWTs, I have delegated this section to another post. If you are not clear on what is in a JWT, how it is created and signed, and how it is verified, then I suggest you take a few minutes and read my post that explains it in detail.

How do I use the `passport-jwt` Strategy??

Before we get into the implementation of the `passport-jwt` strategy, I wanted to make a few notes about implementing JWTs in an authentication strategy.

Unfortunately and fortunately, there are many ways that you can successfully implement JWTs into your application. Because of this, if you search Google for "How to implement JWT in an Express App", you'll get a variety of implementations. Let's take a look at our options from most complex to least complex.

Most Complex: If we wanted to make this process as complicated (but also as transparent) as possible, we could use the signing and verifying process that we used earlier in this post using the built-in Node `crypto` library. This would require us to write a lot of Express middleware, a lot of custom logic, and a lot of error handling, but it could certainly be done.

Somewhat Complex: If we wanted to simplify things a little bit, we could do everything on our own, but instead of using the built-in Node `crypto` library, we could abstract away a lot of complexity and use the popular package `jsonwebtoken`. This is not a terrible idea, and there are actually many tutorials online that show you how to implement JWT authentication using just this library.

Simple (if used correctly): Last but not least, we could abstract away even more complexity and use the `passport-jwt` strategy. Or wait… Don't we need the `passport-local` strategy too since we are authenticating with usernames and passwords? And how do we generate a JWT in the first place? Clearly we will need the `jsonwebtoken` library to do this…

And here lies the problem.

The `passport-jwt` strategy does not have much documentation, and I personally believe that because of this, the questions I just raised create a world of confusion in the development community. This results in hundreds of different implementations of `passport-jwt` combined with external libraries, custom middlewares, and much more. This could be considered a good thing, but for someone looking to implement `passport-jwt` the "correct way", it can be frustrating.

Like any software package, if you use it correctly, it will add value to your development. If you use it incorrectly, it could introduce more complexity to your project than if you never used it in the first place.

In this section, I will do my best to explain what the `passport-jwt` strategy aims to achieve and how we can use it in a way that actually adds *value* to our codebase rather than *complexity*.

So let me start by conveying one very important fact about `passport-jwt`.

The Passport JWT strategy uses the `jsonwebtoken` library.

Why is this important??

Remember–JWTs need to first be *signed* and then *verified*. Passport takes care of the *verification* for us, so we just need to sign our JWTs and send them off to the `passport-jwt` middleware to be verified. Since `passport-jwt` uses the `jsonwebtoken` library to verify tokens, then we should probably be using the same library to *generate* the tokens!

In other words, we need to get familiar with the `jsonwebtoken` library, which begs the question… Why do we even need Passport in the first place??

With the `passport-local` strategy, Passport was useful to us because it connected seamlessly with `express-session` and helped manage our user session. If we wanted to authenticate a user, we use the `passport.authenticate()` method on the `/login` POST route.

```
router.post('/login', passport.authenticate('local', {}), (req, res,
next) => {
    // If we make it here, our user has been authenticate and has been
attached
    // to the current session
});
```

If we wanted to authenticate a route (after the user had logged in), all we needed to do was this:

```
router.get('/protected', (req, res, next) => {
    if (req.isAuthenticated()) {
        // Send the route data
        res.status(200).send('Web page data');
    } else {
        // Not authorized
        res.status(401).send('You are not authorized to view this');
    }
});
```

We were able to do this (after the user had logged in) because the `passport-local` middleware stored our user in the Express Session. To me, this is a bit odd, because you are only using the `passport.authenticate()` method one time (for login).

Now that we are using JWTs, we need to authenticate **every single request**, and thus, we will be using the `passport.authenticate()` method a lot more.

The basic flow looks like this:

1. User logs in with username and password
2. Express server validates the username and password, signs a JWT, and sends that JWT back to the user.
3. The user will store the JWT in the browser (this is where our Angular app comes in) via `localStorage`.
4. For every request, Angular will add the JWT stored in `localStorage` to the `Authorization` HTTP Header (similar to how we stored our session in the `Cookie` header)
5. For every request, the Express app will run the `passport.authenticate()` middleware, which will extract the JWT from the `Authorization` header, verify it with a Public Key, and based on the result, either allow or disallow a user from visiting a route or making an API call.

In summary, to authenticate using the `passport-jwt` strategy, our routes will look like so:

```
/**
 * Session is set to false because we are using JWTs, and don't need a
session! * If you do not set this to false, the Passport framework will
try and
 * implement a session
 */
router.get('/protected', passport.authenticate('jwt', { session: false
}), (req, res, next) => {
    res.status(200).send('If you get this data, you have been
authenticated via JWT!');
});
```

All we need to do is configure Passport with our public/private keys, desired JWT algorithm (RSA256 in our case), and a verify function.

Yes, we could implement our own `passport.authenticate()` middleware, but if we did, we would need to write functions (and error handling… ughhh) to do the following:

- Parse the HTTP header
- Extract the JWT from the HTTP header
- Verify the JWT with `jsonwebtoken`

I would much rather delegate that work (and error handling) to a trusted framework like Passport!

Intro to `jsonwebtoken` and `passport-jwt` configuration

This section will highlight the basic methods and setup of both the `jsonwebtoken` and `passport-jwt` modules irrespective of our Express app. The next section will show how these integrate into the Express and Angular applications.

First, let's see how we could use `jsonwebtoken` to sign and verify a JWT. For this, we will use the same JWT that we used to demonstrate how JWTs worked (below).

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWU
wp5AokiRbz3_oB4OxG-
W9KcEEbDRcZc0nH3L7LzYptiy1PtAylQGxHTWZXtGz4ht0bAecBgmpdgXMguEIcoqPJ1n3pIWk_dUZegpqx0Lka21H6XxUTxiy8Ocaar
GvpCSbr8G8y_Mllj8f4x9nBH8pQux89_6gUY618iYv7tuPWBFfEbLxtF2pZS6YC1aSfLQxeNe8djT9YjpvRZA
```

And here is a basic script that demonstrates how we would sign this JWT and verify it.

So how does `jsonwebtoken` and `passport-jwt` work together? Let's take a look at the configuration for Passport below.

Note on options: The way that options are assigned in the `passport-jwt` library can be a bit confusing. You can pass `jsonwebtoken` options, but they must be passed in a specific way. Below is an object with ALL possible options you can use for your `passport-jwt` object. I left out the `secretOrKeyProvider` option because it is the alternative to the `secretOrKey` option, which is more common. The `secretOrKeyProvider` is a callback function used to retrieve a asymmetric key from a `jwks` key provider. For explanation of any of these options, you can see the passport-jwt docs, this rfc and the jsonwebtoken documentation.

passportJWTOptions.js

The above code (before the options) does the following:

1. When a user visits a protected route, they will attach their JWT to the HTTP `Authorization` header
2. `passport-jwt` will grab that value and parse it using the `ExtractJwt.fromAuthHeaderAsBearerToken()` method.
3. `passport-jwt` will take the extracted JWT along with the options we set and call the `jsonwebtoken` library's `verify()` method.
4. If the verification is successful, `passport-jwt` will find the user in the database, attach it to the `req` object, and allow the user to visit the given resource.

What about Angular? How does that handle JWTs?

If you remember from part 1 of this post, HTTP `Cookies` are automatically sent with every HTTP request (until they expire) after the `Set-Cookie` HTTP header has set the value of them. With JWTs, this is not the case!

We have two options:

1. We can "intercept" each HTTP request from our Angular application and append the `Authorization` HTTP Header with our JWT token
2. We can manually add our JWT token to each request

Yes, the first option is a little bit of up-front work, but I think we can manage it.

In addition to the problem of the JWT not being added to each request automatically, we also have the problem of Angular routing. Since Angular runs in the browser and is a Single Page Application, it is not making an HTTP request every time it loads a new view/route. Unlike a standard Express application where you actually get the HTML from the Express app itself, Angular delivers the HTML all at once, and then the client-side logic determines how the routing works.

Because of this, we are going to need to build an Authentication Service in our Angular application that will keep track of our user's authentication state. We will then allow the user to visit protected Angular routes based on this state.

So if we back up for a second, there are really two layers of authentication going on right now. On one hand, we have the authentication that happens on the Express server, which determines what HTTP requests our user can make. Since we are using Angular as a front-end, all of the HTTP requests that we make to our Express app will be data retrieval. On the other hand, we have authentication within our Angular app. We could just ignore this authentication completely, but what if we had an Angular component view that loaded data from the database?

If the user is logged out on the Express side of things, this component view will try to load data to display, but since the user is not authenticated on the backend, the data request will fail, and our view will look weird since there is no data to display.

A better way to handle this is by synchronizing the two authentication states. If the user is not authorized to make a particular GET request for data, then we should probably not let them visit the Angular route that displays that data. They won't be able to see the data no matter what, but this behaviour creates a much more seamless and friendly user experience.

Below is the code that we will use for our AuthService and Interceptor. For now, don't worry about how this integrates into the Angular application as I will show that later in the implementation section.

authInterceptor.ts

I suggest reading through all the comments to better understand how each service is working.

You can think of the HTTP Interceptor as "middleware" for Angular. It will take the existing HTTP request, add the `Authorization` HTTP header with the JWT stored in `localStorage`, and call the `next()` "middleware" in the chain.

And that's it. We are ready to build this thing.

JWT Based Authentication Implementation

It is finally time to jump into the actual implementation of JWT Authentication with an Express/Angular application. Since we have already covered a lot of the ExpressJS basics (middleware, cookies, sessions, etc.), I will not be devoting sections here to them, but I will briefly walk through some of the Angular concepts. If anything in this application doesn't make sense, be sure to read the first half of this post.

All of the code below can be found in this example repository on Github on the `final` branch.

Initial Setup (skim this section)

Let's first take a very quick glance at the starting code (file names commented at top of each code snippet):

app.js

The only slightly irregular thing above is the database connection. Many times, you will see the connection being made from within `app.js`, but I did this to highlight that the `mongoose.connection` object is global. You can configure it in one module and use it freely in another. By calling `require('./config/database');`, we are creating that global object. The file that defines the `User` model for the database is `./models/user.js`.

```
// File: ./models/user.jsconst mongoose = require('mongoose');const
UserSchema = new mongoose.Schema({
    username: String,
    hash: String,
    salt: String
});mongoose.model('User', UserSchema);
```

Next, we have the routes.

/routes/index.js

/routes/users.js

Finally, we have an entire Angular app in the `angular/` directory. I generated this using the `ng new` command. The only tweaks made to this so far are in `./angular/angular.json`.

```
// File: ./angular/angular.json
..."outputPath": "../public", // Line 16...
```

In the first file, we need to set the output directory so that the `ng build` command builds our Angular application to the `./public/` directory that our Express app serves static content from.

API Routes

Our first step is to write the logic around password validation. To keep things consistent, I will be using the *exact same logic* as I did with the Session Based Authentication example in the first half of this post.

Let's make a folder `./lib` and place a `utils.js` file in it.

utils.js

The above is the same exact module that we used before. Now, let's create routes that will allow us to register a user and login.

users.js

Using Postman (or another HTTP request utility), test the route and create a user. Here is my post request:

```
{
    "username": "zach",
    "password": "123"
}
```

And the results…

```
{
    "success": true,
    "user": {
        "_id": "5def83773d50a20d27887032",
        "username": "zach",
        "hash":
"9aa8c8999e4c25880aa0f3b1b1ae6fbcfdfdedb9fd96295e370a4ecb4e9d30f83d5d91e86d840cc5323e7c4ed15097db5c2262a
        "salt":
"d63bb43fc411a55f0ac6ff8c145c58f70c8c10e18915b5c6d9578b997d637143",
        "__v": 0
    }
}
```

We now have a user in the database that we can test our authentication on, but we currently do not have any logic to use for the `/login` route. This is where Passport comes in.

Add `passport.js` to the `./config/` directory and put the following in it.

passport.js

This is the function that will run on every route that we use the `passport.authenticate()` middleware. Internally, Passport will verify the supplied JWT with the `jsonwebtoken` verify method.

Next, let's create a utility function that will generate a JWT for our user, and put it in the `utils.js` file.

Add this to utils.js

Finally, let's implement the `/users/login/` route so that if the user logs in successfully, they will receive a JWT token in the response.

Add to users.js

Time to try it out! In Postman, make send a POST request to `/users/login/` with the following data (remember, we already created a user):

```
{
    "username": "zach",
    "password": "123"
}
```

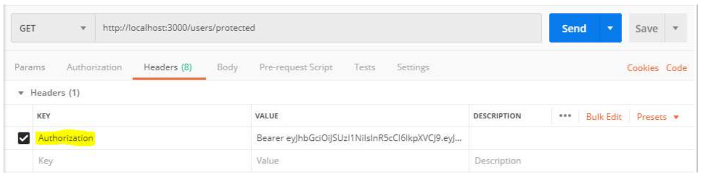
When you send that request, you should get the following result (your JWT will be different because you are using a different private key to sign it):

```
{
    "success": true,
    "token": "Bearer
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI1ZGVmODM3NzNkNTBhMjBkMjc4ODcwMzIiLCJpYXQiOjE1NzYxMTc4NDA
wt7etD94qH_C_rxL745reGMOrtJNy2SffAlAmhcphs4xlbGRjtBoABxHfiL0Hhht2fbGCwf79s5gDlTC9WqWMq8gcXZkLYXnRQZcHCOv
yar_c6cNVxFJBU6ah2sK1mUPTR6ReXUWt_A1lu2aOtgUG-
9wXVp9h3Lh3LrdHuTqF4oV2vbTSMGCzAs33C1wwjdCGqCj3dkqfMSE43f7SSAy2-
m6TgPAPm0QEUV8PiEpS1GlUCsBKVeVYC5hbUyUDS3PaJYQxklIHVNGNqlyj_1IdNaCuquGvyQDDyflZpJKnUPg1WZVgkDa5hVZerrb8h
rt3cWUlVItmJsT30sUInDRsfAevDX83gEtD2QR4ZkZA8ppb9s7Yi6V2_L7JUz5aBPUYT4YQo0iNj4_jpaZByqdp03GFGbfv4tmk-
oeYnJHwgntoBWk_hfE3h5GbCmtfmlTO5A4CWAMu5W5pNanjNsVzogXrUZCfNaY42HC24blpO507-
Vo-GwdIpFCMnrgCLa6DAW3XH-ePlRL-cbIv0-
QFiSCge2RerWx5d3qlD9yintqmXf1TyzB3X7IM_JbVYqVB0sGAPrFBZqk0q0",
    "expiresIn": "1d"
}
```

We will now try this out using a brand new route. In the `./routes/users.js` file, add the following route:

```
router.get('/protected', passport.authenticate('jwt', { session: false
}), (req, res, next) => {
    res.status(200).json({ success: true, msg: "You are successfully
authenticated to this route!"});
});
```

Now in Postman, copy the JWT token you received into the `Authorization` HTTP header.



http://localhost:3000/users/protected

	KEY	VALUE	DESCRIPTION
✓	Authorization	Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ...	
	Key	Value	Description

Add the JWT to the Authorization HTTP Header

When you send this request, you should get the expected response of "Your JWT is valid". If you don't get this request, check your files with mine stored at this Github repo.

Now that your backend is working correctly, it is time to implement the Angular side of things. First, generate the following components:

```
ng generate component register
ng generate component login
ng generate component protected-component
```

Let's get these components and the Angular router setup. Below are the files you will need to update with comments in them explaining some of the logic.

app.module.ts

app.component.html

And now for each component:

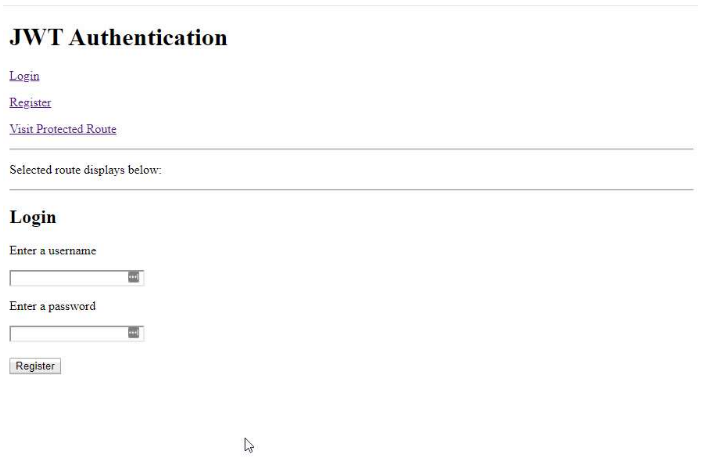
login.component.html

login.component.ts

register.component.html

register.component.ts

If all goes well, your app should look something like this:



Basic working Angular App

Now comes the part where we actually implement our JWT authentication. The first thing we need to wire up is the ability to send POST requests from our login and register routes.

First, we need to add the `HttpClientModule` to our app. In `./angular/src/app/app.module.ts`, add the following import.

```
import { HttpClientModule } from '@angular/common/http';...imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes, {useHash: true}),
    HttpClientModule
],...
```

Now, we can use this in our other components. Update `./angular/src/app/register/register.component.ts` with the following:

You can now visit the register component and register yourself on the Express application. Add the same logic to the login component.

login.component.ts

Finally, let's add some logic to the protected route. In this route, we will make a GET request to our `/users/protected` route, which should return a `401 Unauthorized` error if our JWT is not valid. Since we haven't written the logic to attach the JWT to each request yet, we should get the error.

In the HTML file of the component, add this one line.

```
<!-- ./angular/src/app/protected-component/protected-component.html -->
<!-- This will print the value of the `message` variable in protected-
component.component.ts -->
<p>Message: {{ message }}</p>
```

And in `./angular/src/app/protected-component.component.ts`, add the logic to handle the HTTP request.

protected-component.component.ts

If you visit the protected route right now, you should get the unauthorized error. But wouldn't it be nice if we were able to successfully get data from this GET request? Let's set up our AuthService. Create the following folder and file, and install the `moment` module:

```
mkdir ./angular/src/app/services
touch ./angular/src/app/services/auth.service.ts
npm install --save moment
```

Now add the following code to your service.

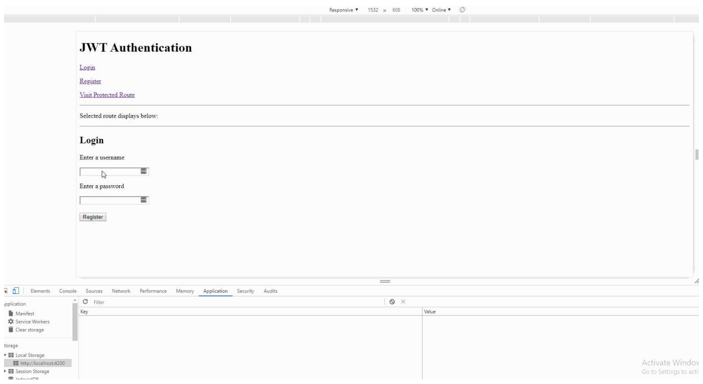
auth.service.ts

In this service, we have methods that will create, read, update, and destroy JWT information stored in the browser's `localStorage` module. The last thing you need to do is add this service to `app.module.ts`.

```
// File: ./angular/src/app/app.module.tsimport { AuthService } from
'./services/auth.service';...providers: [
    AuthService
],...
```

We now need to add some functionality to the `login.component.ts` to set the JWT that we receive after logging in to `localStorage`.

After adding this, you should be able to login and have the JWT saved to `localStorage`.



JWT being issued

Now that we are saving the JWT to `localStorage` after logging in, the only step left is to implement our HTTP interceptor that will retrieve the JWT sitting in `localStorage` and attach it to the HTTP `Authorization` header on every request!

Make the following folder and file.

```
mkdir ./angular/src/app/interceptors
touch ./angular/src/app/interceptors/auth-interceptor.ts
```

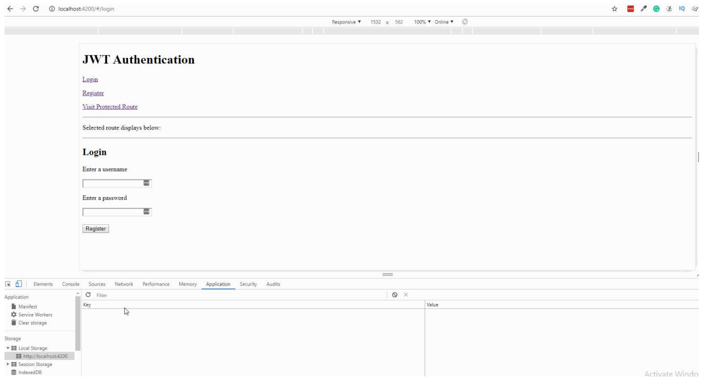
Add the following to this file:

auth-interceptor.ts

And finally, you will need to import it to `app.module.ts`.

```
import { AuthInterceptor } from './interceptors/auth-
interceptor';...providers: [
    AuthService,
    {
      provide: HTTP_INTERCEPTORS,
      useClass: AuthInterceptor,
      multi: true
    }
],
```

And with that, all of your HTTP requests should get the `Authorization` HTTP header populated with a JWT (if it exists in localStorage) on every request!



Completed application

Conclusion

You now have a skeleton application to work with and implement in whatever way you like! I recommend adding additional features like an AuthGuard to handle route authentication even further, but what I have shown you here should get you more than started!

If you have any questions or notice any errors in the post, please let me know in the comments below.