Zach Gollwitzer
Posted on Oct 18, 2020 • Updated on Oct 20, 2020 • Originally published at zachgollwitzer.com

The Ultimate Guide to Passport JS

#javascript #node #passportjs #expressjs

This post also can be viewed as a YouTube series here.

In this post, I am going to walk through why the `Passport-JWT` authentication strategy is a simple, secure solution for small teams and startups implementing a Node/Express + Angular web app.

To understand why a JWT authentication flow is the best choice for this situation, I am going to take you through what authentication options are available to you, how they work, and how to implement them (with the exclusion of OAuth as this is out of scope).

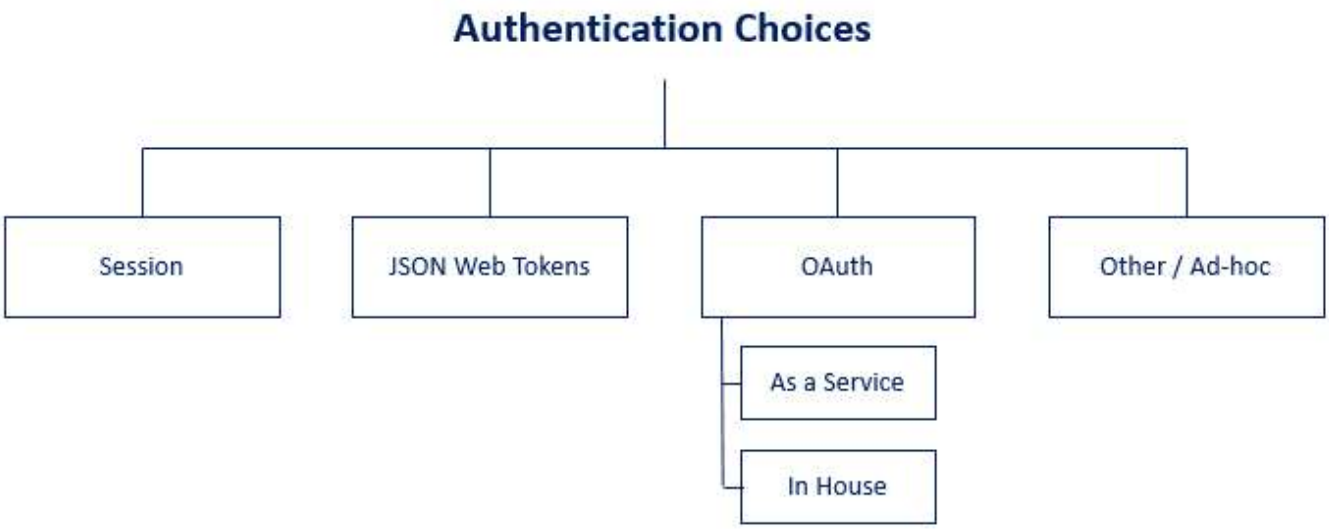
Since this post is long and detailed, if you are already familiar with a topic discussed, just skip it. Likewise, if you are just looking for instructions on how to implement a particular authentication method, you can jump to those sections below:

- Session Based Authentication Implementation
- JWT Based Authentication Implementation

Also, I created sample applications utilizing both authentication methods in the following repositories:

- Session Based Auth Repo
- JWT Auth Repo

Authentication Choices



Above is a high-level overview of the main authentication choices available to developers today. Here is a quick overview of each:

- Session Based Authentication - Utilizes browser Cookies along with backend "Sessions" to manage logged-in and logged-out users.
- JWT Authentication - A stateless authentication method where a JSON Web token (JWT) is stored in the browser (usually `localStorage`). This JWT has assertions about a user and can only be decoded using a secret that is stored on the server.
- OAuth and OpenID Connect Authentication - A modern authentication method where an application uses "claims" generated from other applications to authenticate its own users. In other words, this is federated authentication where an existing service (like Google) handles the authentication and storage of users while your application leverages this flow to authenticate users.

One note I'll make--Oauth can become confusing really quickly, and therefore is not fully explored in this post. Not only is it unnecessary for a small team/startup getting an

application off the ground, but it is also highly variable depending on which service you are using (i.e. Google, Facebook, Github, etc.).

Finally, you might notice that OAuth is listed as "As a service" and "In house". This is a specific note made to highlight the fact that there is actually a company called "OAuth" that implements the OAuth protocol... As a service. You can implement the OAuth protocol without using OAuth the company's service!

What is Session-Based Authentication?

If we were to create a lineage for these authentication methods, session-based authentication would be the oldest of them all, but certainly not obsolete. This method of authentication is "server-side", which means our Express application and database work together to keep the current authentication status of each user that visits our application.

To understand the basic tenets of session-based-authentication, you need to understand a few concepts:

- Basic HTTP Header Protocol
- What a cookie is
- What a session is
- How the session (server) and cookie (browser) interact to authenticate a user

HTTP Headers

There are many ways to make an HTTP request in a browser. An HTTP client could be a web application, IoT device, command line (curl), or a multitude of others. Each of these clients connect to the internet and make HTTP requests which either fetch data (GET), or modify data (POST, PUT, DELETE, etc.).

For explanation purposes, let's assume that:

Server = www.google.com
Client = random guy in a coffee shop working on a laptop

When that random person from the coffee shop types `www.google.com` into their Google Chrome browser, this request will be sent with "HTTP Headers". These HTTP Headers are key:value pairs that provide additional data to the browser to help complete the request. This request will have two types of headers:

1. General Headers
2. Request Headers

To make this interactive, open Google Chrome, open your developer tools (right click, "Inspect"), and click on the "Network" tab. Now, type `www.google.com` into your address bar, and watch as the Network tab loads several resources from the server. You should see several columns like Name, Status, Type, Initiator, Size, Time, and Waterfall. Find the request that has "document" as the "Type" value and click on it. You should see all the headers for this request and response interaction.

The request that you (as the client) made will have General and Request headers resembling (but not exact) the following:

```
General Headers
  Request URL: https://www.google.com/
  Request Method: GET
  Status Code: 200

Request Headers
  Accept: text/html
  Accept-Language: en-US
  Connection: keep-alive
```

When you typed `www.google.com` into your address bar and pressed enter, your HTTP request was sent with these headers (and probably a few others). Although these headers are relatively self-explanatory, I want to walk through a few to get a better idea of what HTTP Headers are used for. Feel free to look up any you don't know on MDN.

The `General` headers can be a mix of both request and response data. Clearly, the `Request URL` and `Request Method` are part of the request object and they tell the Google Chrome browser where to route your request. The `Status Code` is clearly part of the response because it indicates that your GET request was successful and the webpage at `www.google.com` loaded okay.

The `Request Headers` only contain headers included with the request object itself. You can think of request headers as "instructions for the server". In this case, my request tells the Google server the following:

- Hey Google Server, please send me HTML or text data. I'm either incapable or not interested in reading anything else right now!
- Hey Google Server, please only send me English words
- Hey Google Server, please don't close my connection with you after the request is over

There are many more request headers that you can set, but these are just a few common ones that you will probably see on all HTTP requests.

So when you searched for `www.google.com`, you sent your request and the headers to the Google Server (for simplicity, we will just assume it is one big server). The Google Server accepted your request, read through the "instructions" (headers), and created a *response*. The response was comprised of:

- HTML data (what you see in your Browser)
- HTTP Headers

As you might have guessed, the "Response Headers" were those set by the Google Server. Here are a few that you might see:

```
Response Headers
  Content-Length: 41485
  Content-Type: text/html; charset=UTF-8
  Set-Cookie: made_up_cookie_name=some value; expires=Thu, 28-Dec-2020 20:44:50 GMT;
```

These response headers are fairly straightforward with the exception of the `Set-Cookie` header.

I included the `Set-Cookie` header because it is exactly what we need to understand in order to learn what Session-based Authentication is all about (and will help us understand other auth methods later in this post).

How Cookies Work

Without Cookies in the browser, we have a problem.

If we have a protected webpage that we want our users to login to access, without cookies, those users would have to login every time they refresh the page! That is because the HTTP protocol is by default "stateless".

Cookies introduce the concept of "persistent state" and allow the browser to "remember" something that the server told it previously.

The Google Server can tell my Google Chrome Browser to give me access to a protected page, but the second I refresh the page, my browser will "forget" this and make me authenticate again.

This is where Cookies come in, and explains what the `Set-Cookie` header is aiming to do. In the above request where we typed in `www.google.com` into our browser and pressed enter, our client sent a request with some headers, and the Google Server responded with a response and some headers. One of these response headers was `Set-Cookie: made_up_cookie_name=some value; expires=Thu, 28-Dec-2020 20:44:50 GMT;`. Here's how this interaction works:

Server: "Hey client! I want you to set a cookie called `made_up_cookie_name` and set it equal to `some value`.

Client: "Hey server, I will set this on the `Cookie` header of all my requests to this domain until Dec 28, 2020!"

We can verify that this actually happened in Google Chrome Developer Tools. Go to "Application"->"Storage" and click "Cookies". Now click on the site that you are currently visiting and you will see all the cookies that have been set for this site. In our made-up example, you might see something like:

Name	**Value**	**Expires / Max-Age**
made_up_cookie_name	some value	2020-12-28T20:44:50.674Z

This cookie will now be set to the `Cookie` **Request Header** on all requests made to `www.google.com` until the expiry date set on the cookie.

As you might conclude, this could be extremely useful for authentication if we set some sort of "auth" cookie. An overly simplified process of how this might work would be:

1. Random person from the coffee shop types `www.example-site.com/login/` into the browser
2. Random person from the coffee shop fills out a form on this page with a username and password

3. Random person's Google Chrome Browser submits a POST request with the login data (username, password) to the server running `www.example-site.com`.
4. The server running `www.example-site.com` receives the login info, checks the database for that login info, validates the login info, and if successful, creates a response that has the header `Set-Cookie: user_is_authenticated=true; expires=Thu, 1-Jan-2020 20:00:00 GMT`.
5. The random person's Google Chrome browser receives this response and sets a browser cookie:

Name	Value	Expires / Max-Age
user_is_authenticated	true	2020-12-28T20:44:50.674Z

1. The random person now visits `www.example-site.com/protected-route/`
2. The random person's browser creates an HTTP request with the header `Cookie: user_is_authenticated=true; expires=Thu, 1-Jan-2020 20:00:00 GMT` attached to the request.
3. The server receives this request, sees that there is a cookie on the request, "remembers" that it had authenticated this user just a few seconds ago, and allows the user to visit the page.

The Reality of this Situation

Obviously, what I have just described would be a highly insecure way to authenticate a user. In reality, the server would create some sort of hash from the password the user-provided, and validate that hash with some crypto library on the server.

That said, the high-level concept is valid, and it allows us to understand the value of cookies when talking about authentication.

Keep this example in mind as we move through the remainder of this post.

Sessions

Sessions and cookies are actually quite similar and can get confused because they can actually be used *together* quite seamlessly. The *main difference* between the two is the **location** of their storage.

In other words, a Cookie is *set* by the server, but stored in the Browser. If the server wants to use this Cookie to store data about a user's "state", it would have to come up with an elaborate scheme to constantly keep track of what the cookie in the browser looks like. It might go something like this:

- Server: Hey browser, I just authenticated this user, so you should store a cookie to remind me (`Set-Cookie: user_auth=true; expires=Thu, 1-Jan-2020 20:00:00 GMT`) next time you request something from me
- Browser: Thanks, server! I will attach this cookie to my `Cookie` request header
- Browser: Hey server, can I see the contents at `www.domain.com/protected`? Here is the cookie you sent me on the last request.
- Server: Sure, I can do that. Here is the page data. I have also included another `Set-Cookie` header (`Set-Cookie: marketing_page_visit_count=1; user_ip=192.1.234.21`) because the company who owns me likes to track how many people have visited this specific page and from which computer for marketing purposes.
- Browser: Okay, I'll add that cookie to my `Cookie` request header
- Browser: Hey Server, can you send me the contents at `www.domain.com/protected/special-offer`? Here are all the cookies that you have set on me so far. (`Cookie: user_auth=true; expires=Thu, 1-Jan-2020 20:00:00 GMT; marketing_page_visit_count=1; user_ip=192.1.234.21`)

As you can see, the more pages the browser visits, the more cookies the Server sets, and the more cookies the Browser must attach in each request Header.

The Server might have some function that parses through all the cookies attached to a request and perform certain actions based on the presence or absence of a specific cookie. To me, this naturally begs the question... Why doesn't the server just keep a record of this information in a database and use a single "session ID" to identify events that a user is taking?

This is exactly what a session is for. As I mentioned, the main difference between a cookie and a session is *where* they are stored. A session is stored in some Data Store (fancy term for a database) while a Cookie is stored in the Browser. Since the session is stored on the server, it can store sensitive information. Storing sensitive information in a cookie would be highly insecure.

Now where this all gets a bit confusing is when we talk about using cookies and session *together*.

Since Cookies are the method in which the client and server communicate metadata (among other HTTP Headers), a session must still utilize cookies. The easiest way to see this interaction is by actually building out a simple authentication application in Node + Express + MongoDB. I will assume that you have a basic understanding of building apps in Express, but I will try to explain each piece as we go.

Setup a basic app:

```
mkdir session-auth-app
cd session-auth-app
npm init -y
npm install --save express mongoose dotenv connect-mongo express-session passport passpor
```

Here is `app.js`. Read through the comments to learn more about what is going on before continuing.

```
const express = require("express");
const mongoose = require("mongoose");
const session = require("express-session");

// Package documentation - https://www.npmjs.com/package/connect-mongo
const MongoStore = require("connect-mongo")(session);

/**
 * -------------- GENERAL SETUP ----------------
 */

// Gives us access to variables set in the .env file via `process.env.VARIABLE_NAME` synt
require("dotenv").config();

// Create the Express application
var app = express();

// Middleware that allows Express to parse through both JSON and x-www-form-urlencoded re
// These are the same as `bodyParser` - you probably would see bodyParser put here in mos
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
```

```
/**
 * -------------- DATABASE ----------------
 */

/**
 * Connect to MongoDB Server using the connection string in the `.env` file.  To implemer
 * string into the `.env` file
 *
 * DB_STRING=mongodb://<user>:<password>@localhost:27017/database_name
 */

const connection = mongoose.createConnection(process.env.DB_STRING);

// Creates simple schema for a User.  The hash and salt are derived from the user's giver
const UserSchema = new mongoose.Schema({
  username: String,
  hash: String,
  salt: String,
});

// Defines the model that we will use in the app
mongoose.model("User", UserSchema);

/**
 * -------------- SESSION SETUP ----------------
 */

/**
 * The MongoStore is used to store session data.  We will learn more about this in the pc
 *
 * Note that the `connection` used for the MongoStore is the same connection that we are
 */
const sessionStore = new MongoStore({
  mongooseConnection: connection,
  collection: "sessions",
});

/**
 * See the documentation for all possible options - https://www.npmjs.com/package/express
 *
 * As a brief overview (we will add more later):
 *
 * secret: This is a random string that will be used to "authenticate" the session.  In a
 * you would want to set this to a long, randomly generated string
 *
```

```
 * resave: when set to true, this will force the session to save even if nothing changed.
 * the app will still run but you will get a warning in the terminal
 *
 * saveUninitialized: Similar to resave, when set true, this forces the session to be sav
 */
app.use(
  session({
    secret: process.env.SECRET,
    resave: false,
    saveUninitialized: true,
    store: sessionStore,
  })
);

/**
 * -------------- ROUTES ----------------
 */

// When you visit http://localhost:3000/login, you will see "Login Page"
app.get("/login", (req, res, next) => {
  res.send("<h1>Login Page</h1>");
});

app.post("/login", (req, res, next) => {});

// When you visit http://localhost:3000/register, you will see "Register Page"
app.get("/register", (req, res, next) => {
  res.send("<h1>Register Page</h1>");
});

app.post("/register", (req, res, next) => {});

/**
 * -------------- SERVER ----------------
 */

// Server listens on http://localhost:3000
app.listen(3000);
```

The first thing we need to do is understand how the `express-session` module is working within this application. This is a "middleware", which is a fancy way of saying that it is a function that modifies something in our application.

Quick Refresher on Express Middleware

Let's say we had the following code:

```
const express = require("express");

var app = express();

// Custom middleware
function myMiddleware1(req, res, next) {
  req.newProperty = "my custom property";
  next();
}

// Another custom middleware
function myMiddleware2(req, res, next) {
  req.newProperty = "updated value";
  next();
}

app.get("/", (req, res, next) => {
  res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});

// Server listens on http://localhost:3000
app.listen(3000);
```

As you can see, this is an extremely simple Express application that defines two middlewares and has a single route that you can visit in your browser at `http://localhost:3000`. If you started this application and visited that route, it would say "Custom Property Value: undefined" because defining middleware functions alone is not enough.

We need to tell the Express application to actually use these middlewares. We can do this in a few ways. First, we can do it within a route.

```
app.get("/", myMiddleware1, (req, res, next) => {
  res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

If you add the first middleware function as an argument to the route, you will now see "Custom Property Value: my custom property" show up in the browser. What really

happened here:

1. The application was initialized
2. A user visited `http://localhost:3000/` in the browser, which triggered the `app.get()` function.
3. The Express application first checked to see if there was any "global" middleware installed on the router, but it didn't find any.
4. The Express application looked at the `app.get()` function and noticed that there was a middleware function installed before the callback. The application ran the middleware and passed the middleware the `req` object, `res` object, and the `next()` callback.
5. The `myMiddleware1` middleware first set `req.newProperty`, and then called `next()`, which tells the Express application "Go to the next middleware". If the middleware did not call `next()`, the browser would get "stuck" and not return anything.
6. The Express app did not see any more middleware, so it continued with the request and sent the result.

This is just one way to use middleware, and it is exactly how the `passport.authenticate()` function (more on this later, so keep in mind) works.

Another way we can use middleware is by setting it "globally". Take a look at our app after this change:

```
const express = require("express");

var app = express();

// Custom middleware
function myMiddleware1(req, res, next) {
  req.newProperty = "my custom property";
  next();
}

// Another custom middleware
function myMiddleware2(req, res, next) {
  req.newProperty = "updated value";
  next();
}

app.use(myMiddleware2);

app.get("/", myMiddleware1, (req, res, next) => {
```

```
  // Sends "Custom Property Value: my custom property
  res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});

// Server listens on http://localhost:3000
app.listen(3000);
```

With this app structure, you will notice that visiting `http://localhost:3000/` in the browser *still* returns the same value as before. This is because the `app.use(myMiddleware2)` middleware is happening *before* the `app.get('/', myMiddleware1)`. If we removed the middleware from the route, you will see the updated value in the browser.

```
app.use(myMiddleware2);

app.get("/", (req, res, next) => {
  // Sends "Custom Property Value: updated value
  res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

We could also get this result by placing the second middleware after the first within the route.

```
app.get("/", myMiddleware1, myMiddleware2, (req, res, next) => {
  // Sends "Custom Property Value: updated value
  res.send(`<h1>Custom Property Value: ${req.newProperty}`);
});
```

Although this is a quick and high-level overview of middleware in Express, it will help us understand what is going on with the `express-session` middleware.

How Express Session Middleware works

As I mentioned before, the `express-session` module gives us middleware that we can use in our application. The middleware is defined in this line:

```
// Again, here is the documentation for this - https://www.npmjs.com/package/express-sess
app.use(
  session({
    secret: process.env.SECRET,
    resave: false,
    saveUninitialized: true,
    store: sessionStore,
```

```
  })
);
```

Here is a brief overview of what the Express Session Middleware is doing:

1. When a route is loaded, the middleware checks to see if there is a session established in the Session Store (MongoDB database in our case since we are using the `connect-mongo` custom Session Store).
2. If there is a session, the middleware validates it cryptographically and then tells the Browser whether the session is valid or not. If it is valid, the Browser automatically attaches the `connect.sid` Cookie to the HTTP request.
3. If there is no session, the middleware creates a new session, takes a cryptographic hash of the session, and stores that value in a Cookie called `connect.sid`. It then attaches the `Set-Cookie` HTTP header to the `res` object with the hashed value (`Set-Cookie: connect.sid=hashed value`).

You might be wondering why this is useful at all, and how all this actually works.

If you remember from the quick refresher on Express Middlewares, I said that a middleware has the ability to alter the `req` and `res` objects that are passed from one middleware to the next until it reaches the end of the HTTP request. Just like we set a custom property on the `req` object, we could also set something much more complex like a `session` object that has properties, methods, etc.

That is exactly what the `express-session` middleware does. When a new session is created, the following properties are added to the `req` object:

- `req.sessionID` - A randomly generated UUID. You can define a custom function to generate this ID by setting the `genid` option. If you do not set this option, the default is to use the `uid-safe` module.

```
app.use(
  session({
    genid: function (req) {
      // Put your UUID implementation here
    },
  })
);
```

- `req.session` - The Session object. This contains information about the session and is available for setting custom properties to use. For example, maybe you want to track how many times a particular page is loaded in a single session:

```
app.get("/tracking-route", (req, res, next) => {
  if (req.session.viewCount) {
    req.session.viewCount = req.session.viewCount + 1;
  } else {
    req.session.viewCount = 1;
  }

  res.send("<p>View count is: " + req.session.viewCount + "</p>");
});
```

- `req.session.cookie` - The Cookie object. This defines the behavior of the cookie that stores the hashed session ID in the browser. Remember, once the cookie has been set, the browser will attach it to every HTTP request automatically until it expires.

How Passport JS Local Strategy works

There is one last thing that we need to learn in order to fully understand Session Based Authentication--Passport JS.

Passport JS has over 500 authentication "Strategies" that can be used within a Node/Express app. Many of these strategies are highly specific (i.e. `passport-amazon` allows you to authenticate into your app via Amazon credentials), but they all work similar within your Express app.

In my opinion, the Passport module could use some work in the department of documentation. Not only does Passport consist of two modules (Passport base + Specific Strategy), but it is also a middleware, which as we saw is a bit confusing in its own right. To add to the confusion, the strategy that we are going to walk through (`passport-local`) is a middleware that modifies an object created by another middleware (`express-session`). Since the Passport documentation has little to say around how this all works, I will attempt to explain it to the best of my ability in this post.

Let's first walk through the setup of the module.

If you have been following along with this tutorial, you already have the modules needed. If not, you will need to install Passport and a Strategy to your project.

```
npm install --save passport passport-local
```

Once you have done that, you will need to implement Passport within your application. Below, I have added all the pieces you need for the `passport-local` strategy. I have removed comments to simplify. Take a quick read through the code and then we will walk through all of the `// NEW` code.

```
const express = require("express");
const mongoose = require("mongoose");
const session = require("express-session");

// NEW
const passport = require("passport");
const LocalStrategy = require("passport-local").Strategy;
var crypto = require("crypto");
// ---

const MongoStore = require("connect-mongo")(session);

require("dotenv").config();

var app = express();

const connection = mongoose.createConnection(process.env.DB_STRING);

const UserSchema = new mongoose.Schema({
  username: String,
  hash: String,
  salt: String,
});

mongoose.model("User", UserSchema);

const sessionStore = new MongoStore({
  mongooseConnection: connection,
  collection: "sessions",
});

app.use(
  session({
    secret: process.env.SECRET,
    resave: false,
    saveUninitialized: true,
```

```
    store: sessionStore,
  })
);

// NEW
// START PASSPORT

function validPassword(password, hash, salt) {
  var hashVerify = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");
  return hash === hashVerify;
}

function genPassword(password) {
  var salt = crypto.randomBytes(32).toString("hex");
  var genHash = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");

  return {
    salt: salt,
    hash: genHash,
  };
}

passport.use(
  new LocalStrategy(function (username, password, cb) {
    User.findOne({ username: username })
      .then((user) => {
        if (!user) {
          return cb(null, false);
        }

        // Function defined at bottom of app.js
        const isValid = validPassword(password, user.hash, user.salt);

        if (isValid) {
          return cb(null, user);
        } else {
          return cb(null, false);
        }
      })
      .catch((err) => {
        cb(err);
```

```
      });
  })
);

passport.serializeUser(function (user, cb) {
  cb(null, user.id);
});

passport.deserializeUser(function (id, cb) {
  User.findById(id, function (err, user) {
    if (err) {
      return cb(err);
    }
    cb(null, user);
  });
});

app.use(passport.initialize());
app.use(passport.session());

// ---
// END PASSPORT

app.get("/login", (req, res, next) => {
  res.send("<h1>Login Page</h1>");
});

app.post("/login", (req, res, next) => {});

app.get("/register", (req, res, next) => {
  res.send("<h1>Register Page</h1>");
});

app.post("/register", (req, res, next) => {});

app.listen(3000);
```

Yes, I know there is a lot to take in here. Let's start with the easy parts--the helper functions. In the code above, I have two helper functions that will assist in creating and validating a password.

```
/**
 *
 * @param {*} password - The plain text password
```

```
 * @param {*} hash - The hash stored in the database
 * @param {*} salt - The salt stored in the database
 *
 * This function uses the crypto library to decrypt the hash using the salt and then comp
 * the decrypted hash/salt with the password that the user provided at login
 */
function validPassword(password, hash, salt) {
  var hashVerify = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");
  return hash === hashVerify;
}

/**
 *
 * @param {*} password - The password string that the user inputs to the password field i
 *
 * This function takes a plain text password and creates a salt and hash out of it.  Inst
 * password in the database, the salt and hash are stored for security
 *
 * ALTERNATIVE: It would also be acceptable to just use a hashing algorithm to make a has
 * You would then store the hashed password in the database and then re-hash it to verify
 */
function genPassword(password) {
  var salt = crypto.randomBytes(32).toString("hex");
  var genHash = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");

  return {
    salt: salt,
    hash: genHash,
  };
}
```

In addition to the comments, I'll note that these functions require the NodeJS built-in `crypto` library. Some would argue a better crypto library, but unless your application requires a high degree of security, this library is plenty sufficient!

Next up, let's take a look at the `passport.use()` method.

```
/**
 * This function is called when the `passport.authenticate()` method is called.
 *
 * If a user is found an validated, a callback is called (`cb(null, user)`) with the user
 * object.  The user object is then serialized with `passport.serializeUser()` and added
 * `req.session.passport` object.
 */
passport.use(
  new LocalStrategy(function (username, password, cb) {
    User.findOne({ username: username })
      .then((user) => {
        if (!user) {
          return cb(null, false);
        }

        // Function defined at bottom of app.js
        const isValid = validPassword(password, user.hash, user.salt);

        if (isValid) {
          return cb(null, user);
        } else {
          return cb(null, false);
        }
      })
      .catch((err) => {
        cb(err);
      });
  })
);
```

I know the above function is quite a lot to look at, so let's explore some of its key components. First, I'll mention that with **all** Passport JS authentication strategies (not just the local strategy we are using), you will need to supply it with a callback that will be executed when you call the `passport.authenticate()` method. For example, you might have a login route in your app:

```
app.post(
  "/login",
  passport.authenticate("local", { failureRedirect: "/login" }),
  (err, req, res, next) => {
    if (err) next(err);
    console.log("You are logged in!");
```

```
  }
);
```

Your user will type in their username and password via a login form, which will create an HTTP POST request to the `/login` route. Let's say your post request contained the following data:

```
{
  "email": "sample@email.com",
  "pw": "sample password"
}
```

This WILL NOT WORK. The reason? Because the `passport.use()` method *expects* your POST request to have the following fields:

```
{
  "username": "sample@email.com",
  "password": "sample password"
}
```

It looks for `username` and `password` field. If you wanted the first json request body to work, you would need to supply the `passport.use()` function with field definitions:

```
passport.use(
  {
    usernameField: "email",
    passwordField: "pw",
  },
  function (email, password, callback) {
    // Implement your callback function here
  }
);
```

By defining the `usernameField` and `passwordField`, you can specify a custom POST request body object.

That aside, let's return to the POST request at the `/login` route:

```
app.post(
  "/login",
  passport.authenticate("local", { failureRedirect: "/login" }),
```

```
  (err, req, res, next) => {
    if (err) next(err);
    console.log("You are logged in!");
  }
);
```

When the user submits his/her login credentials, the `passport.authenticate()` method (used as middleware here) will execute the callback that you have defined and supply it with the `username` and `password` from the POST request body. The `passport.authenticate()` method takes two parameters--the name of the strategy, and options. The default strategy name here is `local`, but you could change this like so:

```
// Supply a name string as the first argument to the passport.use() function
passport.use("custom-name", new Strategy());

// Use the same name as above
app.post(
  "/login",
  passport.authenticate("custom-name", { failureRedirect: "/login" }),
  (err, req, res, next) => {
    if (err) next(err);
    console.log("You are logged in!");
  }
);
```

The way I have used the `passport.authenticate()` strategy will first execute the callback function that we defined within `new LocalStrategy()`, and if the authentication is successful, it will call the `next()` function, and we will enter the route. If authentication was not successful (invalid username or password), the app will redirect to the `/login` route again.

Now that we understand how it is used, let's return to the callback function that we defined earlier and that `passport.authenticate()` is using.

```
// Tells Passport to use this strategy for the passport.authenticate() method
passport.use(
  new LocalStrategy(
    // Here is the function that is supplied with the username and password field from th
    function (username, password, cb) {
      // Search the MongoDB database for the user with the supplied username
      User.findOne({ username: username })
```

```
        .then((user) => {
          /**
           * The callback function expects two values:
           *
           * 1. Err
           * 2. User
           *
           * If we don't find a user in the database, that doesn't mean there is an appli
           * so we use `null` for the error value, and `false` for the user value
           */
          if (!user) {
            return cb(null, false);
          }

          /**
           * Since the function hasn't returned, we know that we have a valid `user` obje
           * validate the `user` object `hash` and `salt` fields with the supplied passwo
           * utility function.  If they match, the `isValid` variable equals True.
           */
          const isValid = validPassword(password, user.hash, user.salt);

          if (isValid) {
            // Since we have a valid user, we want to return no err and the user object
            return cb(null, user);
          } else {
            // Since we have an invalid user, we want to return no err and no user
            return cb(null, false);
          }
        })
        .catch((err) => {
          // This is an application error, so we need to populate the callback `err` fiel
          cb(err);
        });
    }
  )
);
```

I have commented the above in great detail, so be sure to read through before moving on.

As you may notice, the callback function is database agnostic and validation agnostic. In other words, we don't need to use MongoDB nor do we need to validate our passwords in

the same way. PassportJS leaves this up to us! This can be confusing, but is also extremely powerful and is why PassportJS has such widespread adoption.

Next, you'll see two related functions:

```
passport.serializeUser(function (user, cb) {
  cb(null, user.id);
});

passport.deserializeUser(function (id, cb) {
  User.findById(id, function (err, user) {
    if (err) {
      return cb(err);
    }
    cb(null, user);
  });
});
```

Personally, I found these two functions to be the most confusing because there is not a lot of documentation around them. We will further explore what these functions are doing when we talk about how PassportJS and Express Session middleware interact, but in short, these two functions are responsible for "serializing" and "deserializing" users to and from the current session object.

Instead of storing the entire `user` object in the session, we only need to store the database ID for the user. When we need to get more information about the user in the current session, we can use the deserialize function to look the user up in the database using the ID that was stored in the session. Again, we will make more sense of this soon.

Finally, with the Passport implementation, you will see two more lines of code:

```
app.use(passport.initialize());
app.use(passport.session());
```

If you remember from earlier in the post on how middleware works, by calling `app.use()`, we are telling Express to execute the functions within the parentheses **in order** on **every request**.

In other words, for every HTTP request our Express app makes, it will execute `passport.initialize()` and `passport.session()`.

Something seem weird here??

If `app.use()` **executes** the function contained within, then the above syntax is like saying:

```
passport.initialize()();
passport.session()();
```

The reason this works is because these two functions actually return another function! Kind of like this:

```
Passport.prototype.initialize = function () {
  // Does something

  return function () {
    // This is what is called by `app.use()`
  };
};
```

This is not necessary to know to use Passport, but definitely clears up some confusion if you were wondering about that syntax.

Anyways...

These two middleware functions are necessary for integrating PassportJS with `express-session` middleware. That is why these two functions **must come AFTER** the `app.use(session({}))` middleware! Just like `passport.serializeUser()` and `passport.deserializeUser()`, these middlewares will make much more sense shortly.

Conceptual Overview of Session Based Authentication

Now that we understand HTTP Headers, Cookies, Middleware, Express Session middleware, and Passport JS middleware, it is finally time to learn how to use these to authenticate users into our application. I want to first use this section to review and explain the conceptual flow, and then dive into the implementation in the next section.

Here is a basic flow of our app:

1. Express app starts and listens on `http://www.expressapp.com` (just assume this is true for the sake of the example).
2. A user visits `http://www.expressapp.com/login` in the browser

3. The `express-session` middleware realizes that there is a user connecting to the Express server. It checks the `Cookie` HTTP header on the `req` object. Since this user is visiting for the first time, there is no value in the `Cookie` header. Because there is no `Cookie` value, the Express server returns the `/login` HTML and calls the `Set-Cookie` HTTP header. The `Set-Cookie` value is the cookie string generated by `express-session` middleware according to the options set by the developer (assume in this case the maxAge value is 10 days).
4. The user realizes that he doesn't want to login right now, but instead, wants to go for a walk. He closes his browser.
5. The user returns from his walk, opens the browser, and returns to `http://www.expressapp.com/login` again.
6. Again, the `express-session` middleware runs on the GET request, checks the `Cookie` HTTP header, but this time, finds a value! This is because the user had previously created a session earlier that day. Since the `maxAge` option was set to 10 days on the `express-session` middleware, closing the browser does not destroy the cookie.
7. The `express-session` middleware now takes the `connect.sid` value from the `Cookie` HTTP header, looks it up in the `MongoStore` (fancy way to say that it looks up the id in the database in the `sessions` collection), and finds it. Since the session exists, the `express-session` middleware does not do anything, and both the `Cookie` HTTP header value and the `MongoStore` database entry in the `sessions` collection stays the same.
8. Now, the user types in his username and password and presses the "Login" button.
9. By pressing the "Login" button, the user sends a POST request to the `/login` route, which uses the `passport.authenticate()` middleware.
10. On every request so far, the `passport.initialize()` and `passport.session()` middlewares have been running. On each request, these middlewares are checking the `req.session` object (created by the `express-session` middleware) for a property called `passport.user` (i.e. `req.session.passport.user`). Since the `passport.authenticate()` method had not been called yet, the `req.session` object did not have a `passport` property. Now that the `passport.authenticate()` method has been called via the POST request to `/login`, Passport will execute our user-defined authentication callback using the username and password our user typed in and submitted.
11. We will assume that the user was already registered in the database and typed in the correct credentials. The Passport callback validates the user successfully.
12. The `passport.authenticate()` method now returns the `user` object that was validated. In addition, it attaches the `req.session.passport` property to the `req.session` object, serializes the user via `passport.serializeUser()`, and attaches the serialized user (i.e. the

ID of the user) to the `req.session.passport.user` property. Finally, it attaches the full user object to `req.user`.

13. The user turns off his computer and goes for another walk because our application is boring.
14. The user turns on his computer the next day and visits a **protected route** on our application.
15. The `express-session` middleware checks the `Cookie` HTTP header on `req`, finds the session from yesterday (still valid since our `maxAge` was set to 10 days), looks it up in `MongoStore`, finds it, and does nothing to the `Cookie` since the session is still valid. The middleware re-initializes the `req.session` object and sets to the value returned from `MongoStore`.
16. The `passport.initialize()` middleware checks the `req.session.passport` property and sees that there is still a `user` value there. The `passport.session()` middleware uses the `user` property found on `req.session.passport.user` to re-initialize the `req.user` object to equal the user attached to the session via the `passport.deserializeUser()` function.
17. The protected route looks to see if `req.session.passport.user` exists. Since the Passport middleware just re-initialized it, it does, and the protected route allows the user access.
18. The user leaves his computer for 2 months.
19. The user comes back and visits the same protected route (hint: the session has expired!)
20. The `express-session` middleware runs, realizes that the value of the `Cookie` HTTP header has an **expired** cookie value, and replaces the `Cookie` value with a new Session via the `Set-Cookie` HTTP header attached to the `res` object.
21. The `passport.initialize()` and `passport.session()` middlewares run, but this time, since `express-session` middleware had to create a new session, there is no longer a `req.session.passport` object!
22. Since the user did not log in and is trying to access a protected route, the route will check if `req.session.passport.user` exists. Since it doesn't, access is denied!
23. Once the user logs in again and triggers the `passport.authenticate()` middleware, the `req.session.passport` object will be re-established, and the user will again be able to visit protected routes.

Phewwww....

Got all that?

Session Based Authentication Implementation

The hard part is over.

Putting everything together, below is your full functional Session Based authentication Express app. Below is the app contained within a single file, but I have also refactored this application closer to what you would use in the real world in this repository.

```
const express = require("express");
const mongoose = require("mongoose");
const session = require("express-session");
var passport = require("passport");
var crypto = require("crypto");
var LocalStrategy = require("passport-local").Strategy;

// Package documentation - https://www.npmjs.com/package/connect-mongo
const MongoStore = require("connect-mongo")(session);

/**
 * -------------- GENERAL SETUP ----------------
 */

// Gives us access to variables set in the .env file via `process.env.VARIABLE_NAME` synt
require("dotenv").config();

// Create the Express application
var app = express();

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

/**
 * -------------- DATABASE ----------------
 */

/**
 * Connect to MongoDB Server using the connection string in the `.env` file.  To implemer
 * string into the `.env` file
 *
 * DB_STRING=mongodb://<user>:<password>@localhost:27017/database_name
 */

const conn = "mongodb://devuser:123@localhost:27017/general_dev";
//process.env.DB_STRING
const connection = mongoose.createConnection(conn, {
  useNewUrlParser: true,
```

```
  useUnifiedTopology: true,
});

// Creates simple schema for a User.  The hash and salt are derived from the user's giver
const UserSchema = new mongoose.Schema({
  username: String,
  hash: String,
  salt: String,
});

const User = connection.model("User", UserSchema);

/**
 * This function is called when the `passport.authenticate()` method is called.
 *
 * If a user is found an validated, a callback is called (`cb(null, user)`) with the user
 * object.  The user object is then serialized with `passport.serializeUser()` and added
 * `req.session.passport` object.
 */
passport.use(
  new LocalStrategy(function (username, password, cb) {
    User.findOne({ username: username })
      .then((user) => {
        if (!user) {
          return cb(null, false);
        }

        // Function defined at bottom of app.js
        const isValid = validPassword(password, user.hash, user.salt);

        if (isValid) {
          return cb(null, user);
        } else {
          return cb(null, false);
        }
      })
      .catch((err) => {
        cb(err);
      });
  })
);

/**
 * This function is used in conjunction with the `passport.authenticate()` method.  See c
 * `passport.use()` above ^^ for explanation
```

```
 */
passport.serializeUser(function (user, cb) {
  cb(null, user.id);
});

/**
 * This function is used in conjunction with the `app.use(passport.session())` middleware
 * Scroll down and read the comments in the PASSPORT AUTHENTICATION section to learn how
 *
 * In summary, this method is "set" on the passport object and is passed the user ID stor
 * object later on.
 */
passport.deserializeUser(function (id, cb) {
  User.findById(id, function (err, user) {
    if (err) {
      return cb(err);
    }
    cb(null, user);
  });
});

/**
 * -------------- SESSION SETUP ----------------
 */

/**
 * The MongoStore is used to store session data.  We will learn more about this in the pc
 *
 * Note that the `connection` used for the MongoStore is the same connection that we are
 */
const sessionStore = new MongoStore({
  mongooseConnection: connection,
  collection: "sessions",
});

/**
 * See the documentation for all possible options - https://www.npmjs.com/package/express
 *
 * As a brief overview (we will add more later):
 *
 * secret: This is a random string that will be used to "authenticate" the session.  In a
 * you would want to set this to a long, randomly generated string
 *
 * resave: when set to true, this will force the session to save even if nothing changed.
 * the app will still run but you will get a warning in the terminal
```

```
 *
 * saveUninitialized: Similar to resave, when set true, this forces the session to be sav
 *
 * store: Sets the MemoryStore to the MongoStore setup earlier in the code.  This makes i
 * saved in a MongoDB database in a "sessions" table and used to lookup sessions
 *
 * cookie: The cookie object has several options, but the most important is the `maxAge`
 * the cookie will expire when you close the browser.  Note that different browsers behav
 * behavior (for example, closing Chrome doesn't always wipe out the cookie since Chrome
 * background and "remember" your last browsing session)
 */
app.use(
  session({
    //secret: process.env.SECRET,
    secret: "some secret",
    resave: false,
    saveUninitialized: true,
    store: sessionStore,
    cookie: {
      maxAge: 1000 * 30,
    },
  })
);

/**
 * -------------- PASSPORT AUTHENTICATION ----------------
 */

/**
 * Notice that these middlewares are initialized after the `express-session` middleware.
 * Passport relies on the `express-session` middleware and must have access to the `req.s
 *
 * passport.initialize() - This creates middleware that runs before every HTTP request.
 *      1. Checks to see if the current session has a `req.session.passport` object on it
 *
 *          { user: '<Mongo DB user ID>' }
 *
 *      2.  If it finds a session with a `req.session.passport` property, it grabs the Us
 *          internal Passport method for later.
 *
 * passport.session() - This calls the Passport Authenticator using the "Session Strategy
 * steps that this method takes:
 *      1.  Takes the MongoDB user ID obtained from the `passport.initialize()` method (r
 *          it to the `passport.deserializeUser()` function (defined above in this module
 *          function will look up the User by the given ID in the database and return it.
```

```
 *      2.  If the `passport.deserializeUser()` returns a user object, this user object i
 *          and can be accessed within the route.  If no user is returned, nothing happer
 */
app.use(passport.initialize());
app.use(passport.session());

/**
 * -------------- ROUTES ----------------
 */

app.get("/", (req, res, next) => {
  res.send("<h1>Home</h1>");
});

// When you visit http://localhost:3000/login, you will see "Login Page"
app.get("/login", (req, res, next) => {
  const form = '<h1>Login Page</h1><form method="POST" action="/login">\
    Enter Username:<br><input type="text" name="username">\
    <br>Enter Password:<br><input type="password" name="password">\
    <br><br><input type="submit" value="Submit"></form>';

  res.send(form);
});

// Since we are using the passport.authenticate() method, we should be redirected no matt
app.post(
  "/login",
  passport.authenticate("local", {
    failureRedirect: "/login-failure",
    successRedirect: "login-success",
  }),
  (err, req, res, next) => {
    if (err) next(err);
  }
);

// When you visit http://localhost:3000/register, you will see "Register Page"
app.get("/register", (req, res, next) => {
  const form = '<h1>Register Page</h1><form method="post" action="register">\
                  Enter Username:<br><input type="text" name="username">\
                  <br>Enter Password:<br><input type="password" name="password">\
                  <br><br><input type="submit" value="Submit"></form>';

  res.send(form);
});
```

```
app.post("/register", (req, res, next) => {
  const saltHash = genPassword(req.body.password);

  const salt = saltHash.salt;
  const hash = saltHash.hash;

  const newUser = new User({
    username: req.body.username,
    hash: hash,
    salt: salt,
  });

  newUser.save().then((user) => {
    console.log(user);
  });

  res.redirect("/login");
});

/**
 * Lookup how to authenticate users on routes with Local Strategy
 * Google Search: "How to use Express Passport Local Strategy"
 *
 * Also, look up what behavior express session has without a max age set
 */
app.get("/protected-route", (req, res, next) => {
  console.log(req.session);
  if (req.isAuthenticated()) {
    res.send("<h1>You are authenticated</h1>");
  } else {
    res.send("<h1>You are not authenticated</h1>");
  }
});

// Visiting this route logs the user out
app.get("/logout", (req, res, next) => {
  req.logout();
  res.redirect("/login");
});

app.get("/login-success", (req, res, next) => {
  console.log(req.session);
  res.send("You successfully logged in.");
});
```

```
app.get("/login-failure", (req, res, next) => {
  res.send("You entered the wrong password.");
});

/**
 * -------------- SERVER ----------------
 */

// Server listens on http://localhost:3000
app.listen(3000);

/**
 * -------------- HELPER FUNCTIONS ----------------
 */

/**
 *
 * @param {*} password - The plain text password
 * @param {*} hash - The hash stored in the database
 * @param {*} salt - The salt stored in the database
 *
 * This function uses the crypto library to decrypt the hash using the salt and then comp
 * the decrypted hash/salt with the password that the user provided at login
 */
function validPassword(password, hash, salt) {
  var hashVerify = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");
  return hash === hashVerify;
}

/**
 *
 * @param {*} password - The password string that the user inputs to the password field i
 *
 * This function takes a plain text password and creates a salt and hash out of it.  Inst
 * password in the database, the salt and hash are stored for security
 *
 * ALTERNATIVE: It would also be acceptable to just use a hashing algorithm to make a has
 * You would then store the hashed password in the database and then re-hash it to verify
 */
function genPassword(password) {
  var salt = crypto.randomBytes(32).toString("hex");
  var genHash = crypto
```

```
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");

  return {
    salt: salt,
    hash: genHash,
  };
}
```

What is JWT Based Authentication?

Before I start getting lost in the details, I must mention that if you read all the previous sections, this section 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.

Additionally, as we walk through the basics of JWT authentication, we will start to understand why JWT auth is far better for Angular front-end applications (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.

Components of a JSON Web Token (JWT)

At the most basic level, a JSON Web Token (JWT) is just a small piece of data that contains information about a user. It contains three parts:

1. Header
2. Payload
3. Signature

Each part is encoded in Base64url format (easier to transport over HTTP protocol than JSON objects).

Here is an example JWT:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwi
```

Notice how there are periods `.` within this text. These periods separate the header from the payload from the signature. Let's isolate the header:

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9
```

Now, let's install the NodeJS `base64url` library and decode this.

```
npm install --save base64url
```

```
# I am running this from Node console

const base64 = require('base64url');

const headerInBase64UrlFormat = 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9';

const decoded = base64.decode(headerInBase64UrlFormat);

console.log(decoded);
```

If we decode the header as shown above, it will give us the following **JSON** object (hence the name, "JSON" Web Token):

```
{
    "alg":"RS256",
    "typ":"JWT"
}
```

We will get to what this means later, but for now, let's decode the payload and the signature using the same method.

```
# I am running this from Node console

const base64 = require('base64url');

const JWT_BASE64_URL = 'eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwi

// Returns an array of strings separated by the period
const jwtParts = JWT_BASE64_URL.split('.');

const headerInBase64UrlFormat = jwtParts[0];
const payloadInBase64UrlFormat = jwtParts[1];
const signatureInBase64UrlFormat = jwtParts[2];

const decodedHeader = base64.decode(headerInBase64UrlFormat);
const decodedPayload = base64.decode(payloadInBase64UrlFormat);
const decodedSignature = base64.decode(signatureInBase64UrlFormat);
```

```
console.log(decodedHeader);
console.log(decodedPayload);
console.log(decodedSignature);
```

The result of the above code will be:

```
# Header
{
    "alg":"RS256",
    "typ":"JWT"
}

# Payload
{
    "sub":"1234567890",
    "name":"John Doe",
    "admin":true,
    "iat":1516239022
}

# Signature
Lots of gibberish like - ��e宿���(�$[����4\e�'
```

For now, ignore the signature part of the JWT. The reason it cannot be decoded into a meaningful JSON object is because it is a bit more complex than the header and payload. We will be exploring this further soon.

Let's walk through the header and payload.

The header has both an `alg` and `typ` property. These are both in the JWT because they represent "instructions" for interpreting that messy signature.

The payload is the simplest part, and is just information about the user that we are authenticating.

- `sub` - An abbreviation for "subject", and usually represents the user ID in the database
- `name` - Just some arbitrary metadata about the user
- `admin` - Some more arbitrary metadata about the user
- `iat` - An abbreviation for "issued at", and represents when this JWT was issued

With JWTs, you might also see the following information in a payload:

- `exp` - An abbreviation for "expiration time", which indicates the time at which this JWT expires
- `iss` - An abbreviation for "issuer", which is often used when a central login server is issuing many JWT tokens (also used heavily in the OAuth protocol)

You can see all of the "standard claims" for the JWT specification at this link.

Creating the signature step by step

Although I told you not to worry about that gibberish we received when we tried to decode the `signature` portion of the JWT, I'm sure it is still bothersome. In this section, we will learn how that works, but **first**, you're going to need to read this article I wrote which explains how Public Key Cryptography works (should take you 10-20 min depending on how familiar you are with the topic). Even if you are familiar with the topic, you should skim the article. This section will make absolutely zero sense if you don't have a solid understanding of public key cryptography.

Anyways...

The signature of a JWT is actually a combination of the `header` and the `payload`. It is created like so (below is pseudocode):

```
// NOTE: This is pseudocode!!

// Copied from the original JWT we are using as an example above
const base64UrlHeader = "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9";
const base64UrlPayload =
  "eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOT

// We take a one-way hash of the header and payload using the SHA256 hashing
// algorithm.  We know to use this algorithm because it was specified in the
// JWT header
const hashedData = sha256hashFunction(base64UrlHeader + "." + base64UrlPayload);

// The issuer (in our case, it will be the Express server) will sign the hashed
// data with its private key
const encryptedData = encryptFunction(issuer_priv_key, hashedData);

const finalSignature = convertToBase64UrlFunction(encryptedData);
```

Even though `sha256hashFunction`, `encryptFunction`, and `convertToBase64UrlFunction` are made up pseudocode, hopefully the above example explains the process of creating the signature adequately.

Now, let's use the NodeJS `crypto` library to actually implement the above pseudocode. Below are the public and private keys that I used to generate this example JWT (which we will need to create and decode the signature of the JWT).

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAnzyis1ZjfNB0bBgKFMSv
vkTtwlvBsaJq7S5wA+kzeVOVpVWwkWdVha4s38XM/pa/yr47av7+z3VTmvDRyAHc
aT92whREFpLv9cj5lTeJSibyr/Mrm/YtjCZVWgaOYIhwrXwKLqPr/11inWsAkfIy
tvHWTxZYEcXLgAXFuUuaS3uF9gEiNQwzGTU1v0FqkqTBr4B8nW3HCN47XUu0t8Y0
e+lf4s4OxQawWD79J9/5d3Ry0vbV3Am1FtGJiJvOwRsIfVChDpYStTcHTCMqtvWb
V6L11BWkpzGXSW4Hv43qa+GSYOD2QU68Mb59oSk2OB+BtOLpJofmbGEGgvmwyCI9
MwIDAQAB
-----END PUBLIC KEY-----


-----BEGIN RSA PRIVATE KEY-----
MIIEogIBAAKCAQEAnzyis1ZjfNB0bBgKFMSvvkTtwlvBsaJq7S5wA+kzeVOVpVWw
kWdVha4s38XM/pa/yr47av7+z3VTmvDRyAHcaT92whREFpLv9cj5lTeJSibyr/Mr
m/YtjCZVWgaOYIhwrXwKLqPr/11inWsAkfIytvHWTxZYEcXLgAXFuUuaS3uF9gEi
NQwzGTU1v0FqkqTBr4B8nW3HCN47XUu0t8Y0e+lf4s4OxQawWD79J9/5d3Ry0vbV
3Am1FtGJiJvOwRsIfVChDpYStTcHTCMqtvWbV6L11BWkpzGXSW4Hv43qa+GSYOD2
QU68Mb59oSk2OB+BtOLpJofmbGEGgvmwyCI9MwIDAQABAoIBACiARq2wkltjtcjs
kFvZ7w1JAORHbEufEO1Eu27zOIlqbgyAcAl7q+/1bip4Z/x1IVES84/yTaM8p0go
amMhvgry/mS8vNi1BN2SAZEnb/7xSxbflb70bX9RHLJqKnp5GZe2jexw+wyXlwaM
+bclUCrh9e1ltH7IvUrRrQnFJfh+is1fRon9Co9Li0GwoN0x0byrrngU8Ak3Y6D9
D8GjQA4Elm94ST3izJv8iCOLSDBmzsPsXfcCUZfmTfZ5DbUDMbMxRnSo3nQeoKGC
0Lj9FkWcfmLcpGlSXTO+Ww1L7EGq+PT3NtRae1FZPwjddQ1/4V905kyQFLamAA5Y
lSpE2wkCgYEAy1OPLQcZt4NQnQzPz2SBJqQN2P5u3vXl+zNVKP8w4eBv0vWuJJF+
hkGNnSxXQrTkvDOIUddSKOzHHgSg4nY6K02ecyT0PPm/UZvtRpWrnBjcEVtHEJNp
bU9pLD5iZ0J9sbzPU/LxPmuAP2Bs8JmTn6aFRspFrP7W0s1Nmk2jsm0CgYEAyH0X
+jpoqxj4efZfkUrg5GbSEhf+dZglf0tTOA5bVg8IYwtmNk/pniLG/zI7c+GlTc9B
BwfMr59EzBq/eFMI7+LgXaVUsM/sS4Ry+yeK6SJx/otIMWtDfqxsLD8CPMCRvecC
2Pip4uSgrl0MOebl9XKp57GoaUWRWRHqwV4Y6h8CgYAZhI4mh4qZtnhKjY4TKDjx
QYufXSdLAi9v3FxmvchDwOgn4L+PRVdMwDNms2bsL0m5uPn104EzM6w1vzz1zwKz
5pTpPI0OjgWN13Tq8+PKvm/4Ga2MjgOgPWQkslulO/oMcXbPwWC3hcRdr9tcQtn9
Imf9n2spL/6EDFId+Hp/7QKBgAqlWdiXsWckdE1Fn91/NGHsc8syKvjjk1onDcw0
NvVi5vcba9oGdElJX3e9mxqUKMrw7msJJv1MX8LWyMQC5L6YNYHDfbPF1q5L4i8j
8mRex97UVokJQRRA452V2vCO6S5ETgpnad36de3MUxHgCOX3qL382Qx9/THVmbma
3YfRAoGAUxL/Eu5yvMK8SAt/dJK6FedngcM3JEFNplmtLYVLWhkIlNRGDwkg3I5K
```

```
y18Ae9n7dHVueyslrb6weq7dTkYDi3iOYRW8HRkIQh06wEdbxt0shTzAJvvCQfrB
jg/3747WSsf/zBTcHihTRBdAv6OmdhV4/dD5YBfLAkLrd+mX7iE=
-----END RSA PRIVATE KEY-----
```

First up, let's create both our header and payload. I will be using the `base64url` library for this, so make sure you have it installed.

```
const base64 = require("base64url");

const headerObj = {
  alg: "RS256",
  typ: "JWT",
};

const payloadObj = {
  sub: "1234567890",
  name: "John Doe",
  admin: true,
  iat: 1516239022,
};

const headerObjString = JSON.stringify(headerObj);
const payloadObjString = JSON.stringify(payloadObj);

const base64UrlHeader = base64(headerObjString);
const base64UrlPayload = base64(payloadObjString);

console.log(base64UrlHeader); // eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9
console.log(base64UrlPayload); // eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWR
```

Boom! You just created the first two parts of the JWT. Now, let's add the creation of the signature to this script. We will need the built-in NodeJS `crypto` library and the private key to do this.

```
const base64 = require("base64url");
const crypto = require("crypto");
const signatureFunction = crypto.createSign("RSA-SHA256");
const fs = require("fs");

const headerObj = {
  alg: "RS256",
```

```
  typ: "JWT",
};

const payloadObj = {
  sub: "1234567890",
  name: "John Doe",
  admin: true,
  iat: 1516239022,
};

const headerObjString = JSON.stringify(headerObj);
const payloadObjString = JSON.stringify(payloadObj);

const base64UrlHeader = base64(headerObjString);
const base64UrlPayload = base64(payloadObjString);

signatureFunction.write(base64UrlHeader + "." + base64UrlPayload);
signatureFunction.end();

// The private key without line breaks
const PRIV_KEY = fs.readFileSync(__dirname + "/id_rsa_priv.pem", "utf8");

// Will sign our data and return Base64 signature (not the same as Base64Url!)
const signatureBase64 = signatureFunction.sign(PRIV_KEY, "base64");

const signatureBase64Url = base64.fromBase64(signatureBase64);

console.log(signatureBase64Url); // POstGetfAytaZS82wHcjoTyoqhMyxXiWdR7Nn7A29DNSl0EiXLdwJ
```

In the above code, I have repeated the previous script that we ran with the logic for creating the signature appended. In this code, we first append the header and the payload (base64url encoded) together by a `.`. We then write those contents into our signature function, which is the built-in NodeJS crypto library's `RSA-SHA256` signature class. Although it sounds complicated, all this tells us is to

1. Use an RSA, standard 4096 bit Public/Private keypair
2. For hashing the `base64Url(header) + '.' + base64Url(payload)`, use the `SHA256` hashing algorithm.

In the JWT header, you will notice that this is indicated by `RS256`, which is just an abbreviated way of saying `RSA-SHA256`.

Once we have written the contents into this function, we need to read the private key we will be signing with from a file. I have stored the private key shown earlier in this post in a file called `id_rsa_priv.pem`, which is located in the current working directory and stored in `.pem` format (pretty standard).

Next, I will "sign" the data, which will first hash the data with the `SHA256` hashing function, and then encrypt the result with the private key.

Finally, since the NodeJS crypto library returns our value in `Base64` format, we need to use the `base64Url` library to convert that from Base64->Base64Url.

Once that's done, you will have a JWT header, payload, and signature that match our original JWT perfectly!

Verifying the signature step by step

In the previous section, we looked at how you would create a JWT signature. In user authentication, the flow looks like this:

1. Server receives login credentials (username, password)
2. Server performs some logic to verify that these credentials are valid
3. If the credentials are valid, the server issues and *signs* a JWT and returns it to the user
4. The user uses the issued JWT to authenticate future requests in the browser

But what happens when the user makes another request to a protected route of your application or a protected API endpoint?

Your user presents the server with a JWT token, but how does your server interpret that token and decide whether the user is valid? Below are the basic steps.

1. Server receives a JWT token
2. Server first checks if the JWT token has an expiry, and if that expiration date has been passed. If so, the server denies access.
3. If the JWT is not expired, the server will first convert the `header` and `payload` from Base64Url->JSON format.
4. Server looks in the `header` of the JWT to find which hashing function and encryption algorithm it needs to decrypt the signature (we will assume that in this example, the JWT uses `RSA-SHA256` as the algorithm.

5. Server uses a `SHA256` hashing function to hash `base64Url(header) + '.' + base64Url(payload)`, which leaves the server with a hash value.
6. Server uses the `Public Key` stored in its filesystem to decrypt the `base64Url(signature)` (remember, private key encrypts, public key decrypts). Since the server is both creating the signatures and verifying them, it should have both the Public and Private key stored in its filesystem. For larger use cases, it would be common to have these duties separated to entirely separate machines.
7. Server compares the values from step 5 and step 6. If they match, this JWT is valid.
8. If the JWT is valid, the server uses the `payload` data to get more information about the user and authenticate that user.

Using the **same JWT** that we have been using throughout this post, here is how this process looks in code:

```
const base64 = require("base64url");
const crypto = require("crypto");
const verifyFunction = crypto.createVerify("RSA-SHA256");
const fs = require("fs");

const JWT =
  "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9l
const PUB_KEY = fs.readFileSync(__dirname + "/id_rsa_pub.pem", "utf8");

const jwtHeader = JWT.split(".")[0];
const jwtPayload = JWT.split(".")[1];
const jwtSignature = JWT.split(".")[2];

verifyFunction.write(jwtHeader + "." + jwtPayload);
verifyFunction.end();

const jwtSignatureBase64 = base64.toBase64(jwtSignature);

const signatureIsValid = verifyFunction.verify(
  PUB_KEY,
  jwtSignatureBase64,
  "base64"
);

console.log(signatureIsValid); // true
```

There are several items worthy of note in this code. First, we take the Base64Url encoded JWT and split it into its 3 parts. We then use the built-in NodeJS `createVerify` function to create a new `Verify` class. Just like the process of creating the signature, we need to pass in the `base64url(header) + '.' + base64url(payload)` into the stream used by the `Verify` crypto class.

The next step is critical--you need to convert the `jwtSignature` from its default encoding Base64Url->Base64. You then need to pass the public key, the Base64 version of the signature, and indicate to NodeJS that you are using Base64. If you do not specify the encoding, it will default to a Buffer and you will always get a false return value.

If all goes well, you should get a true return value, which means this signature is valid!

Zoom Out: The true value of JWT signatures

If you read the above two sections, you know how to create and verify a JWT signature using the `RSA-SHA256` JWT algorithm (other algorithms work very similarly, but this algorithm is considered one of the more secure and "production-ready" algorithms).

But what does it all mean?

I know we have gone in all sorts of directions in this post about user authentication, but all of this knowledge comes together here. If you think about authenticating a user with Cookies and Sessions, you know that in order to do so, your application server must have a database keeping track of the sessions, and this database must be called each time a user wants to visit a protected resource on the server.

With JWT authentication, the only thing needed to verify that a user is authenticated is a public key!!

Once a JWT token has been issued (by either your application server, an authentication server, or even a 3rd party authentication server), that JWT can be stored in the browser securely and can be used to verify any request without using a database at all. The application server just needs the public key of the issuer!

If you extrapolate this concept and think about the wider implications of JWT, it becomes clear how powerful it is. You no longer need a local database. You can transport authentication all over the web!

Let's say I log in to a popular service like Google and I receive a JWT token from Google's authentication server. The only thing that is needed to verify the JWT that I am browsing with is the public key that matches the private key Google signed with. Usually, this public key is *publicly* available, which means that anyone on the internet can verify my JWT! If they trust Google and they trust that Google is providing the correct public key, then there is no reason that I cannot just use the JWT issued by Google to authenticate users into **my application**.

I know I said that we wouldn't be getting into all the OAuth stuff in this post, but this is the essence of delegated authentication (i.e. the OAuth2.0 protocol)!

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 thousands 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
 * 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.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwi
```

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

```
const jwt = require("jsonwebtoken");
const fs = require("fs");
```

```
const PUB_KEY = fs.readFileSync(__dirname + "/id_rsa_pub.pem", "utf8");
const PRIV_KEY = fs.readFileSync(__dirname + "/id_rsa_priv.pem", "utf8");

// ================================================================
// ------------------  SIGN -----------------------------------
// ================================================================

const payloadObj = {
  sub: "1234567890",
  name: "John Doe",
  admin: true,
  iat: 1516239022,
};

/**
 * Couple things here:
 *
 * First, we do not need to pass in the `header` to the function, because the
 * jsonwebtoken module will automatically generate the header based on the algorithm spec
 *
 * Second, we can pass in a plain Javascript object because the jsonwebtoken library will
 * pass it into JSON.stringify()
 */
const signedJWT = jwt.sign(payloadObj, PRIV_KEY, { algorithm: "RS256" });

console.log(signedJWT); // Should get the same exact token that we had in our example

// ================================================================
// ------------------  VERIFY ---------------------------------
// ================================================================

// Verify the token we just signed using the public key.  Also validates our algorithm RS
jwt.verify(signedJWT, PUB_KEY, { algorithms: ["RS256"] }, (err, payload) => {
  if (err.name === "TokenExpiredError") {
    console.log("Whoops, your token has expired!");
  }

  if (err.name === "JsonWebTokenError") {
    console.log("That JWT is malformed!");
  }

  if (err === null) {
    console.log("Your JWT was successfully validated!");
  }
```

```
  // Both should be the same
  console.log(payload);
  console.log(payloadObj);
});
```

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

```
const JwtStrategy = require("passport-jwt").Strategy;
const ExtractJwt = require("passport-jwt").ExtractJwt;

const PUB_KEY = fs.readFileSync(__dirname + "/id_rsa_pub.pem", "utf8");

// At a minimum, you must pass these options (see note after this code snippet for more)
const options = {
  jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
  secretOrKey: PUB_KEY,
};

// The JWT payload is passed into the verify callback
passport.use(
  new JwtStrategy(options, function (jwt_payload, done) {
    // We will assign the `sub` property on the JWT to the database ID of user
    User.findOne({ id: jwt_payload.sub }, function (err, user) {
      // This flow look familiar?  It is the same as when we implemented
      // the `passport-local` strategy
      if (err) {
        return done(err, false);
      }
      if (user) {
        return done(null, user);
      } else {
        return done(null, false);
      }
    });
  })
);
```

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.

```
const passportJWTOptions = {
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    secretOrKey: PUB_KEY || secret phrase,
    issuer: 'enter issuer here',
    audience: 'enter audience here',
    algorithms: ['RS256'],
    ignoreExpiration: false,
    passReqToCallback: false,
    jsonWebTokenOptions: {
        complete: false,
        clockTolerance: '',
        maxAge: '2d', // 2 days
        clockTimestamp: '100',
        nonce: 'string here for OpenID'
    }
}
```

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 behavior creates a much more seamless and friendly user experience.

Below is the code that we will use for our AuthService and Interceptor. I found this code in a blog post at Angular University and thought it was extremely simple and clean, so we

will use it here. For now, don't worry about how this integrates into the Angular application as I will show that later in the implementation section.

```
// https://momentjs.com/
import * as moment from "moment";

@Injectable()
export class AuthService {

    /**
     * Gives us access to the Angular HTTP client so we can make requests to
     * our Express app
     */
    constructor(private http: HttpClient) {}

    /**
     * Passes the username and password that the user typed into the application
     * and sends a POST request to our Express server login route, which will
     * authenticate the credentials and return a JWT token if they are valid
     *
     * The `res` object (has our JWT in it) is passed to the setLocalStorage
     * method below
     *
     * shareReplay() documentation - https://www.learnrxjs.io/operators/multicasting/shar
     */
    login(email:string, password:string ) {
        return this.http.post<User>('/users/login', {email, password})
            .do(res => this.setLocalStorage)
            .shareReplay();
    }

    /**
     *
     */
    private setLocalStorage(authResult) {

        // Takes the JWT expiresIn value and add that number of seconds
        // to the current "moment" in time to get an expiry date
        const expiresAt = moment().add(authResult.expiresIn,'second');

        // Stores our JWT token and its expiry date in localStorage
        localStorage.setItem('id_token', authResult.idToken);
        localStorage.setItem("expires_at", JSON.stringify(expiresAt.valueOf()) );
    }
```

```
    // By removing the token from localStorage, we have essentially "lost" our
    // JWT in space and will need to re-authenticate with the Express app to get
    // another one.
    logout() {
        localStorage.removeItem("id_token");
        localStorage.removeItem("expires_at");
    }

    // Returns true as long as the current time is less than the expiry date
    public isLoggedIn() {
        return moment().isBefore(this.getExpiration());
    }

    isLoggedOut() {
        return !this.isLoggedIn();
    }

    getExpiration() {
        const expiration = localStorage.getItem("expires_at");
        const expiresAt = JSON.parse(expiration);
        return moment(expiresAt);
    }
}
```

```
// Note: We will eventually incorporate this into our app.module.ts so that it
// automatically works on all HTTP requests

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const idToken = localStorage.getItem("id_token");

    if (idToken) {
      const cloned = req.clone({
        headers: req.headers.set("Authorization", "Bearer " + idToken),
      });

      return next.handle(cloned);
    } else {
```

```
        return next.handle(req);
      }
    }
}
```

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](#).

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):

```
// File: app.js

const express = require("express");
const cors = require("cors");
const path = require("path");

/**
 * -------------- GENERAL SETUP ----------------
 */

// Gives us access to variables set in the .env file via `process.env.VARIABLE_NAME` syn
require("dotenv").config();
```

```
// Create the Express application
var app = express();

// Configures the database and opens a global connection that can be used in any module w
require("./config/database");

// Must first load the models
require("./models/user");

// Instead of using body-parser middleware, use the new Express implementation of the sam
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

// Allows our Angular application to make HTTP requests to Express application
app.use(cors());

// Where Angular builds to - In the ./angular/angular.json file, you will find this confi
// at the property: projects.angular.architect.build.options.outputPath
// When you run `ng build`, the output will go to the ./public directory
app.use(express.static(path.join(__dirname, "public")));

/**
 * -------------- ROUTES ----------------
 */

// Imports all of the routes from ./routes/index.js
app.use(require("./routes"));

/**
 * -------------- SERVER ----------------
 */

// Server listens on http://localhost:3000
app.listen(3000);
```

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.js

const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
  username: String,
  hash: String,
  salt: String,
});

mongoose.model("User", UserSchema);
```

Next, we have the routes.

```
// File: ./routes/index.js

const router = require("express").Router();

// Use the routes defined in `./users.js` for all activity to http://localhost:3000/users
router.use("/users", require("./users"));

module.exports = router;
```

```
// File: ./routes/users.js

const mongoose = require("mongoose");
const router = require("express").Router();
const User = mongoose.model("User");

// http://localhost:3000/users/login
router.post("/login", function (req, res, next) {});

// http://localhost:3000/users/register
router.post("/register", function (req, res, next) {});

module.exports = router;
```

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.

```
// File: ./lib/util.js

const crypto = require("crypto");

/**
 * -------------- HELPER FUNCTIONS ----------------
 */

/**
 *
 * @param {*} password - The plain text password
 * @param {*} hash - The hash stored in the database
 * @param {*} salt - The salt stored in the database
 *
 * This function uses the crypto library to decrypt the hash using the salt and then comp
 * the decrypted hash/salt with the password that the user provided at login
 */
function validPassword(password, hash, salt) {
  var hashVerify = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");
  return hash === hashVerify;
}
```

```
/**
 *
 * @param {*} password - The password string that the user inputs to the password field i
 *
 * This function takes a plain text password and creates a salt and hash out of it.  Inst
 * password in the database, the salt and hash are stored for security
 *
 * ALTERNATIVE: It would also be acceptable to just use a hashing algorithm to make a has
 * You would then store the hashed password in the database and then re-hash it to verify
 */
function genPassword(password) {
  var salt = crypto.randomBytes(32).toString("hex");
  var genHash = crypto
    .pbkdf2Sync(password, salt, 10000, 64, "sha512")
    .toString("hex");

  return {
    salt: salt,
    hash: genHash,
  };
}

module.exports.validPassword = validPassword;
module.exports.genPassword = genPassword;
```

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.

```
// File: ./routes/users.js

const mongoose = require("mongoose");
const router = require("express").Router();
const User = mongoose.model("User");
const utils = require("../lib/utils");

// http://localhost:3000/users/login
router.post("/login", function (req, res, next) {});

router.post("/register", function (req, res, next) {
  const saltHash = utils.genPassword(req.body.password);

  const salt = saltHash.salt;
  const hash = saltHash.hash;
```

```
  const newUser = new User({
    username: req.body.username,
    hash: hash,
    salt: salt,
  });

  try {
    newUser.save().then((user) => {
      res.json({ success: true, user: user });
    });
  } catch (err) {
    res.json({ success: false, msg: err });
  }
});

module.exports = router;
```

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

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

```
{
    "success": true,
    "user": {
        "_id": "5def83773d50a20d27887032",
        "username": "zach",
        "hash": "9aa8c8999e4c25880aa0f3b1b1ae6fbcfdfdedb9fd96295e370a4ecb4e9d30f83d5d91e8
        "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.

```
// File: ./config/passport

const JwtStrategy = require("passport-jwt").Strategy;
const ExtractJwt = require("passport-jwt").ExtractJwt;
const fs = require("fs");
const path = require("path");
const User = require("mongoose").model("User");

// Go up one directory, then look for file name
const pathToKey = path.join(__dirname, "..", "id_rsa_pub.pem");

// The verifying public key
const PUB_KEY = fs.readFileSync(pathToKey, "utf8");

// At a minimum, you must pass the `jwtFromRequest` and `secretOrKey` properties
const options = {
  jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
  secretOrKey: PUB_KEY,
  algorithms: ["RS256"],
};

// app.js will pass the global passport object here, and this function will configure it
module.exports = (passport) => {
  // The JWT payload is passed into the verify callback
  passport.use(
    new JwtStrategy(options, function (jwt_payload, done) {
      // Since we are here, the JWT is valid!

      // We will assign the `sub` property on the JWT to the database ID of user
      User.findOne({ _id: jwt_payload.sub }, function (err, user) {
        // This flow look familiar?  It is the same as when we implemented
        // the `passport-local` strategy
        if (err) {
          return done(err, false);
        }
        if (user) {
          // Since we are here, the JWT is valid and our user is valid, so we are authori
          return done(null, user);
        } else {
          return done(null, false);
        }
      });
```

```
    })
  );
};
```

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.

```
// File: ./lib/utils.js

const jsonwebtoken = require("jsonwebtoken");

/**
 * @param {*} user - The user object.  We need this to set the JWT `sub` payload property
 */
function issueJWT(user) {
  const _id = user._id;

  const expiresIn = "1d";

  const payload = {
    sub: _id,
    iat: Date.now(),
  };

  const signedToken = jsonwebtoken.sign(payload, PRIV_KEY, {
    expiresIn: expiresIn,
    algorithm: "RS256",
  });

  return {
    token: "Bearer " + signedToken,
    expires: expiresIn,
  };
}
```

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.

```
// File: ./routes/users.js
const mongoose = require("mongoose");
const router = require("express").Router();
const User = mongoose.model("User");
const passport = require("passport");
const utils = require("../lib/utils");

// Validate an existing user and issue a JWT
router.post("/login", function (req, res, next) {
  User.findOne({ username: req.body.username })
    .then((user) => {
      if (!user) {
        res.status(401).json({ success: false, msg: "could not find user" });
      }

      // Function defined at bottom of app.js
      const isValid = utils.validPassword(
        req.body.password,
        user.hash,
        user.salt
      );

      if (isValid) {
        const tokenObject = utils.issueJWT(user);

        res.status(200).json({
          success: true,
          token: tokenObject.token,
          expiresIn: tokenObject.expires,
        });
      } else {
        res
          .status(401)
          .json({ success: false, msg: "you entered the wrong password" });
      }
    })
    .catch((err) => {
      next(err);
    });
});
```

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.eyJzdWIiOiI1ZGVmODM3NzNkNTBhMjE
    "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.

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.

```
// File: ./angular/src/app/app.module.ts

import { BrowserModule } from "@angular/platform-browser";

// These two modules will help us with Angular forms and submitting data to
// our Express backend
import { NgModule } from "@angular/core";
import { FormsModule } from "@angular/forms";

// This will allow us to navigate between our components
import { Routes, RouterModule } from "@angular/router";

// These are the four components in our app so far
import { AppComponent } from "./app.component";
import { LoginComponent } from "./login/login.component";
import { RegisterComponent } from "./register/register.component";
import { ProtectedComponentComponent } from "./protected-component/protected-component.cc

// Define which route will load which component
const appRoutes: Routes = [
  { path: "login", component: LoginComponent },
  { path: "register", component: RegisterComponent },
  { path: "protected", component: ProtectedComponentComponent },
];

// Your standard Angular setup
@NgModule({
  declarations: [
    AppComponent,
    LoginComponent,
    RegisterComponent,
    ProtectedComponentComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    RouterModule.forRoot(appRoutes, { useHash: true }),
```

```
  ],
  providers: [],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

```
<!-- File: ./angular/src/app/app.component.html -->

<h1>JWT Authentication</h1>

<!-- By clicking these, the component assigned to each route will load below -->
<p><a routerLink="/login">Login</a></p>
<p><a routerLink="/register">Register</a></p>
<p><a routerLink="/protected">Visit Protected Route</a></p>

<hr />
<p>Selected route displays below:</p>
<hr />

<!-- This will load the current route -->
<router-outlet></router-outlet>
```

And now for each component:

```
<!-- File: ./angular/src/app/login/login.component.html -->

<h2>Login</h2>

<form (ngSubmit)="onLoginSubmit()" #loginform="ngForm">
  <div>
    <p>Enter a username</p>
    <input type="text" name="username" ngModel />
    <p>Enter a password</p>
    <input type="password" name="password" ngModel />
  </div>
  <button style="margin-top: 20px;" type="submit">Register</button>
</form>
```

```
// File: ./angular/src/app/login/login.component.ts

import { Component, OnInit, ViewChild } from "@angular/core";
import { NgForm } from "@angular/forms";
```

```
@Component({
  selector: "app-login",
  templateUrl: "./login.component.html",
  styleUrls: ["./login.component.css"],
})
export class LoginComponent implements OnInit {
  // This will give us access to the form
  @ViewChild("loginform", { static: false }) loginForm: NgForm;

  constructor() {}

  // When you submit the form, the username and password values will print to the screen
  onLoginSubmit() {
    console.log(this.loginForm.value.username);
    console.log(this.loginForm.value.password);
  }

  ngOnInit() {}
}
```

```
<!-- File: ./angular/src/app/register/register.component.html -->

<h2>Register</h2>

<form (ngSubmit)="onRegisterSubmit()" #registerform="ngForm">
  <div>
    <p>Enter a username</p>
    <input type="text" name="username" ngModel />
    <p>Enter a password</p>
    <input type="password" name="password" ngModel />
  </div>
  <button style="margin-top: 20px;" type="submit">Register</button>
</form>


// File: ./angular/src/app/register/register.component.ts

import { Component, OnInit, ViewChild } from "@angular/core";
import { NgForm } from "@angular/forms";

@Component({
  selector: "app-register",
```

```
  templateUrl: "./register.component.html",
  styleUrls: ["./register.component.css"],
})
export class RegisterComponent implements OnInit {
  @ViewChild("registerform", { static: false }) registerForm: NgForm;

  constructor() {}

  ngOnInit() {}

  onRegisterSubmit() {
    console.log(this.registerForm.value.username);
    console.log(this.registerForm.value.password);
  }
}
```

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

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:

```
// File: ./angular/src/app/register/register.component.ts
```

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { NgForm } from '@angular/forms';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {

  @ViewChild('registerform', { static: false }) registerForm: NgForm;

  constructor(private http: HttpClient) { }

  ngOnInit() {
  }

  // Submits a post request to the /users/register route of our Express app
  onRegisterSubmit() {
    const username = this.registerForm.value.username;
    const password = this.registerForm.value.password;

    const headers = new HttpHeaders({'Content-type': 'application/json'});

    const reqObject = {
      username: username,
      password: password
    };

    this.http.post('http://localhost:3000/users/register', reqObject, { headers: headers

      // The response data
      (response) => {
        console.log(response);
      },

      // If there is an error
      (error) => {
        console.log(error);
      },

      // When observable completes
      () => {
        console.log('done!');
```

```
      }

    );
  }

}
```

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

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { NgForm } from '@angular/forms';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {

  @ViewChild('loginform', { static: false }) loginForm: NgForm;

  constructor(private http: HttpClient) { }

  onLoginSubmit() {
    const username = this.loginForm.value.username;
    const password = this.loginForm.value.password;

    const headers = new HttpHeaders({'Content-type': 'application/json'});

    const reqObject = {
      username: username,
      password: password
    };

    this.http.post('http://localhost:3000/users/login', reqObject, { headers: headers }).

      // The response data
      (response) => {
        console.log(response);
      },
```

```
      // If there is an error
      (error) => {
        console.log(error);
      },

      // When observable completes
      () => {
        console.log('done!');
      }

    );
  }

  ngOnInit() {
  }

}
```

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
<p>Message: {{ message }}</p>
```

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

```
// File: ./angular/src/app/protected-component.component.ts

import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Component({
  selector: 'app-protected-component',
```

```
  templateUrl: './protected-component.component.html',
  styleUrls: ['./protected-component.component.css']
})
export class ProtectedComponentComponent implements OnInit {

  constructor(private http: HttpClient) { }

  message: String

  // Execute this HTTP request when the route loads
  ngOnInit() {
    this.http.get('http://localhost:3000/users/protected').subscribe(
      (response) => {
        if (response) {
          this.message = 'You are authenticated!';
        }
      },

      (error) => {
        if (error.status === 401) {
          this.message = 'You are not authorized to visit this route.  No data is displa
        }
      },

      () => {
        console.log('HTTP request done');
      }
    );
  }

}
```

If you visit the protected route right now, you should get an 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.

```
// File: ./angular/src/app/services/auth.service.ts

import { Injectable } from '@angular/core';
import * as moment from "moment";

@Injectable()
export class AuthService {

    constructor() {}

    setLocalStorage(responseObj) {
        const expiresAt = moment().add(responseObj.expiresIn);

        localStorage.setItem('id_token', responseObj.token);
        localStorage.setItem("expires_at", JSON.stringify(expiresAt.valueOf()) );
    }

    logout() {
        localStorage.removeItem("id_token");
        localStorage.removeItem("expires_at");
    }

    public isLoggedIn() {
        return moment().isBefore(this.getExpiration());
    }

    isLoggedOut() {
        return !this.isLoggedIn();
    }

    getExpiration() {
        const expiration = localStorage.getItem("expires_at");
        const expiresAt = JSON.parse(expiration);
        return moment(expiresAt);
    }
}
```

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.ts
```

```
import { 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`.

```
// File: ./angular/src/app/login/login.component.ts

// Import auth service
import { AuthService } from '../services/auth.service';

...

// Add service to module
constructor(private http: HttpClient, private authService: AuthService) { }

...

// In post request, when you receive the JWT, use the service to add it to storage
this.http.post('http://localhost:3000/users/login', reqObject, { headers: headers }).subs

  // The response data
  (response) => {

    // If the user authenticates successfully, we need to store the JWT returned in local
    this.authService.setLocalStorage(response);

  },

...
```

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

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:

```
import { Injectable } from "@angular/core";
import {
  HttpRequest,
  HttpHandler,
  HttpEvent,
  HttpInterceptor,
} from "@angular/common/http";
import { Observable } from "rxjs";

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {
    const idToken = localStorage.getItem("id_token");

    if (idToken) {
      const cloned = req.clone({
        headers: req.headers.set("Authorization", idToken),
      });

      return next.handle(cloned);
    } else {
      return next.handle(req);
    }
  }
}
```

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!

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 this massive post, please let me know in the comments below.

Discussion (3)

Will G • Oct 19 '20

This is one hell of a post! I'm looking into implmenting auth on a side project but didn't know where to start. This post was very helpful for my understanding as a beginner! Thank you very much.

Feril Sunu • Jun 13 '21

Well explained! Expecting more posts like this! :)

Aris Zagakos • Aug 2 '21

Excellent work!

Code of Conduct • Report abuse

Zach Gollwitzer

Just trying to write cleaner code each day | My story - https://youtu.be/Zr73KfbiSu0

LOCATION
Cleveland, Ohio

EDUCATION
Bachelor in Corporate Finance, Self Taught Dev

JOINED
Feb 20, 2020

More from Zach Gollwitzer

Lesson 6 - JavaScript Built-In Functions and Objects #fullstackroadmap
#webdev #codenewbie #javascript

Lesson 5 - JavaScript Functions, Loops, and Conditionals #fullstackroadmap
#webdev #javascript #codenewbie

Lesson 4 - JavaScript Operators (Fullstack developer roadmap series)
#codenewbie #webdev #javascript