01/31/2019 • 14 minutes to read

In this article

June 2016

Volume 31 Number 6

[The Working Programmer]

How To Be MEAN: Passport

By Ted Neward | June 2016

Welcome back, MEANers.

I've been doing a ton of server-side work and it's getting close to the time that I start moving over to the client end of things. Before I do that, though, there's one more thing that absolutely needs discussing before I can make the transition entirely. Specifically, I need to be able to support users. Most applications (if not all of them, by this point) require some kind of user authentication mechanism to establish a user's identity, typically so that you can restrict the data that you show them or the options that you allow them to do within the system.

While it's always tempting to "roll your own," within the Node community, that's just so 2010! The right answer to any of these kinds of dilemmas is always "go to npm," and in this case, the widespread hands-down winner around authentication systems is a Node.js library called Passport.

Passport

By this point, it should be straightforward to figure out what the first steps are for using the Passport library; find out the npm package name and "npm install." The npm package name can be discovered either by searching the online npm registry, or by visiting the Passport homepage. However, visiting PassportJS.org first yields two interesting tidbits: one, that contrary to every other Node.js package homepage ever written, the "npm install" command isn't present right there on the front page; two, that Passport apparently has this concept of "strategies," and it's important.

The reason for this is simple: When you say, "It's time to authenticate the user's credentials," that's actually a vague statement. Not only is there a variety of different credentials that might be used to authenticate, there's a thousand or more different credential stores (a la servers) against which a user might authenticate. Passport wants to be the solution to any sort of authentication against any kind of credential store—Facebook, LinkedIn, Google or your own local database—and uses a variety of different kinds of credentials, from username/password through JSON Web Tokens to HTTP Bearer headers and just about anything else that you might dream up.

This means, then, that Passport isn't just one package; there's a core passport package and then there are strategies (307 of them, in fact, at the time of this writing) for how Passport is to do the actual work of authenticating. The choice of strategy (or strategies—I'll get to that in a moment) defines the actual package required, which in turn defines what to install. (But, truth-in-advertising time here, Passport does in fact define a core package "passport" that will be used by the other strategies involved, so you can get a jump on things by doing an "npm install --save passport" before I dive into the strategy details).

Hello, Local

Far and away the most common strategy (particularly for systems that are being built against internal user databases/credential stores) is the "local" strategy. This is the classic, "Client sends a username and a password, and you compare it against ... well, whatever you store usernames and passwords in." It's arguably also not nearly as secure as some of the other strategies, but it's a good place to start.

Right now, in the code I've been working with, there's been no authentication whatsoever. So, let's keep things simple by just hardcoding a fixed username/password in place. Once you see how Passport works, it's relatively easy to see where the code for a database lookup would go to do the comparison, so I'm going to leave that out.

Having decided that I want to use the passport-local strategy, I begin with "npm install --save passport-local" to get the necessary passport bits in place. (Remember, the "--save" argument puts it into the package manifest file so that it'll get automatically tracked as a formal dependency.)

Once installed, I need to do three things: one, configure Passport to use the given strategy; two, establish the HTTP URL route to which the user will be sending the authentication request; three, set up the Express middleware to require authentication before allowing the user to actually access the HTTP URL in question.

Configuration

I'll start by getting Passport loaded up in the application first. Assuming that passport and passport-local have already been installed, I need to load them into the app.js script via the usual require magic:

JavaScript Copy

```javascript
var express = require('express'),
  bodyParser = require('body-parser'),
  // ...
  passport = require('passport'),
  LocalStrategy = require('passport-local').Strategy;
```

Notice that the LocalStrategy is set slightly different; like with MongoClient before, you actually assign LocalStrategy the result of accessing the field "Strategy" out of the object that's returned from the require call. This is'nt common in Node.js, but it's not so rare as to be unique. LocalStrategy in this case is going to serve as a kind of class to be instantiated (or as close to it as JavaScript can generally get).

I also need to tell the Express environment that Passport is on the job:

JavaScript Copy

```javascript
var app = express();
app.use(bodyParser.json());
app.use(passport.initialize());
```

The initialize call is fairly self-explanatory; it will prep Passport to prepare to receive incoming requests. Often, there'll be a similar call to passport.session to set up per-user sessions, similar to what you see in ASP.NET, but for an HTTP API like what I'm building here, that's less often necessary or desirable (I'll talk about that in a bit).

Challenge

Next, I need to establish the callback that Passport will invoke when it receives an authentication request. This callback will do the work of looking up the user and validating the password passed in. (Or, in a more real-world scenario, looking up the user and validating that the salted password hash is the same as the salted password hash currently stored in the database—but that's really more outside the scope of Passport itself.) That's done by calling passport.use and passing in an instance of the Strategy to use, with the callback embedded within it, as shown in **Figure 1**.

Figure 1 Establishing the Callback

JavaScript Copy

```javascript
passport.use(new LocalStrategy(
  function(username, password, done) {
    debug("Authenticating ",username,",",password);
    if ((username === "sa") && (password == "nopassword")) {
      var user = {
        username : "ted",
        firstName : "Ted",
        lastName : "Neward",
        id : 1
      };
      return done(null, user);
    }
    else {
      return done(null, false, { message: "DENIED"} );
    }
  }
));
```

Several things are going on here. First, by the time the callback is invoked, Passport has already done the work of parsing the incoming request and extracting the username and password to pass in to this callback. For the LocalStrategy, Passport assumes that those values are passed in via parameters named username and password, respectively. (This is configurable in the LocalStrategy construction call, if those aren't acceptable.)

Second, the actual mechanism of verification is entirely outside of Passport's jurisdiction; it assumes the strategies will do the verification, and in this case, the "local" strategy defers that entirely to the application code. In this example, you just check against a hardcoded value, but in more conventional cases this would be a Mongo lookup for a user whose username matched what was passed in and then a check against the password.

Third, in keeping with the usual Node.js middleware style, success or failure is signaled by use of the done function, with the parameters passed indicating whether success or failure took place. Success means the second parameter is a user object that will be placed within the Express request object that's passed further on down the pipeline; failure will cause Passport to ask Express to return a 401 (Not Authorized) response, and can optionally include the failure message, usually used for "flash" messages against the UI. (If flash messages aren't being used, the message is effectively thrown away.)

Consequences

Now, all that remains is to configure the route by which the authentication will take place:

JavaScript Copy

```javascript
app.post('/login',
  passport.authenticate('local', { session: false }),
  function(req, res) {
    debug("user ", req.user.firstName, " authenticated against the system");
    res.redirect("/persons");
  });
```

Passport doesn't particularly care what the URL pattern is that does the actual authentication; /login is just a convention, but /signin or /user/auth or any of a half-dozen other varieties would be entirely reasonable. The key is that the first step when resolving this route is to call the passport authenticate function, passing in which strategy to use (local), whether to use per-user session cookies (which, as already noted, is not particularly appropriate for an API), and the actual function to invoke if the authentication succeeds. Here, that function simply logs a message to debug and then redirects the user to the list of Persons stored in the database.

Now, I can test this by passing in either form-POSTed content or by sending in JSON content; because this is an API, it's probably better and easier to send in a JSON packet:

JavaScript Copy

```javascript
{ "username" : "sa" , "password" : "nopassword" }
```

If the username and password match, success and a 302 redirect to /persons is returned; if not, then a 401 response is handed back. It works!

Redirecting Traffic

In fact, it's a common pattern (when building a traditional server-side Web app using Express) that a successful authentication will take the user to a given route, whereas a failure should take the user to a new page, and for this reason, Passport allows for a simpler approach to authenticate's callbacks:

JavaScript Copy

```
app.post('/login',
  passport.authenticate('local', { successRedirect: '/',
                                   failureRedirect: '/login',
                                   failureFlash: true })
);
```

Here, on success, Passport will automatically redirect to the "/" URL, and on failure, back to the "/login" URL, and (in this case) with a flash message indicating that the user failed to sign in successfully.

In the case of an API, though, it's more common to hand back a JSON representation of the user object to the client for display and editing. Bear in mind, however, that nothing security-related or sensitive should ever be sent back as part of this—no passwords, in particular. The browsers are all extremely helpful in providing client-side debugging utilities, and as a result, any attacker could very easily reach into that user object held in the browser's memory and start editing away to their heart's content. That could be bad. (These JSON objects can also be tampered with "in flight," prompting most Node.js-based API systems to run over HTTPS, rather than HTTP. Fortunately, most of the time, configuring Express to run over HTTPS instead of HTTP is more an exercise in cloud configuration than any programmatic change.) As a result, passwords should never leave the server, and "roles" (for a role-based authorization system) should always be checked from the database, not from the user object the request passed in.

As written, however, right now the API client will need to pass authentication credentials each time the "/login" route is hit, and the credentials aren't checked on any of the other routes. While I certainly could put authentication checks on every route (and should, come to think of it), I probably don't want to have to pass the credentials as part of every method call.

Alternatives

Passport has this idea covered "in spades," as they say.

First, you can always go back to turning sessions on; when sessions are on, Passport will create a unique identifier and hand it back as part of the HTTP response as a cookie. Clients are then required to hand that cookie back as part of each subsequent request. The main requirement at that point on the server end is that the Passport library needs to know how to transform a user object into an identifier, and back again; they call this serializing and deserializing a user, and it requires setting up method callbacks for each of these two Passport endpoints:

JavaScript Copy

```javascript
passport.serializeUser(function(user, done) {
  done(null, user.id);
});
passport.deserializeUser(function(id, done) {
  User.findById(id, function(err, user) {
    done(err, user);
  });
});
```

The serializeUser function is designed to provide a unique identifier for the user to Passport (so I grab it out of the user.id field) and the deserializeUser function does the reverse (so I use the id passed in as the primary key in a database lookup for the user object as a whole).

You can turn on sessions for most, if not all, Passport strategies, but in general it works when the server is generating HTML to be interpreted directly by the browser. APIs tend not to work with cookies nearly as much, particularly because APIs are often hit by native mobile app clients as much as, or more often than, a browser-based client.

A second approach uses a different Passport strategy that relies on a “known secret” to both client and server. This can then be passed in a variety of ways. In some cases, the system maintains a known set of issued “API keys,” and you must provide that key as part of each request. This is quite common with a number of third-party REST services, but it bears a serious weakness in that if an attacker can obtain the key, the attacker can masquerade as the client until the client resets the key. Passport provides a strategy for this; use “npm install --save passport-localapikey.” It behaves much the same way as the Local strategy, except now the strategy authentication method will look up the API key in the database, rather than the username and password.

A similar approach makes use of JSON Web Tokens (JWTs), which are more secure, but require a much longer space to explain than what I have here; “npm install --save passport-jwt” brings it into the project. JWTs are a packed set of a variety of different data elements, one of which can be a shared secret (à la API key or password), but can be verified against particular issuers, audience and more.

Or, perhaps, the goal is to not store any sort of credentials at all, but rely on third-party systems (like Facebook, Google, Twitter, LinkedIn or any of several hundred other popular sites) to do the authenticating. Passport has it covered here, as well, with specific strategies for each of these sites individually, as well as generalized OAuth 2.0 (and OpenID, for those sites that use that) strategies.

I think the point is becoming clear: If you can imagine an authentication system, Passport has a strategy already defined for it. Just “npm install,” set up the configuration, put the authorize call in the Express routes and off you go.

By the way, it seems important to point out that there are services across the Internet that will provide a single point of access control for all of these authentication issues. These “Authentication-as-a-Service” services are becoming more popular as the number of sites that people use on a regular basis proliferate and become more and more of an administrative headache. One of my favorites, Auth0 (which actually has a few ex-Microsoft folks in the technical side of the company), is a sponsor for the Passport project, and its icons and logos appear discreetly scattered throughout the Passport site. I would strongly encourage checking it out if the project doesn’t already have a pre-determined authentication strategy in place (such as a legacy system or integrating against Facebook or Dropbox, or what have you).

Wrapping Up

Passport is arguably the most successful authentication projects ever developed, across any language or platform. It manages to provide the necessary authentication “hooks” while leaving open the actual means of authentication when you want to control that, yet slipping in and doing all that heavy lifting when you don’t. The strategy approach means it’s infinitely extensible, and can accommodate any sort of new authentication scheme that might emerge, even 20 years into the future. (Don’t laugh—all this JavaScript will, in fact, still be running 20 years from now. You watch.)

But Passport is defined almost as much by what it doesn’t do as what it does; it completely punts on any idea of role-based authorization and it doesn’t try to address any kind of encryption or cryptography. Passport is all about credentials checking, which of course by this point makes the name a lot clearer—just as when I travel to Europe, I need to show my passport to prove that I am an American citizen, Passport requires that users display their credentials so that they can prove they are citizens in good standing within the system.

Once again, I find myself out of space and time, so for now...happy coding!

Ted Neward *is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP and has authored and coauthored a dozen books. Reach him at ted@tedneward.com. if you’re interested in having him come work with your team, or read his blog at blogs.tedneward.com.*

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth

Discuss this article in the MSDN Magazine forum