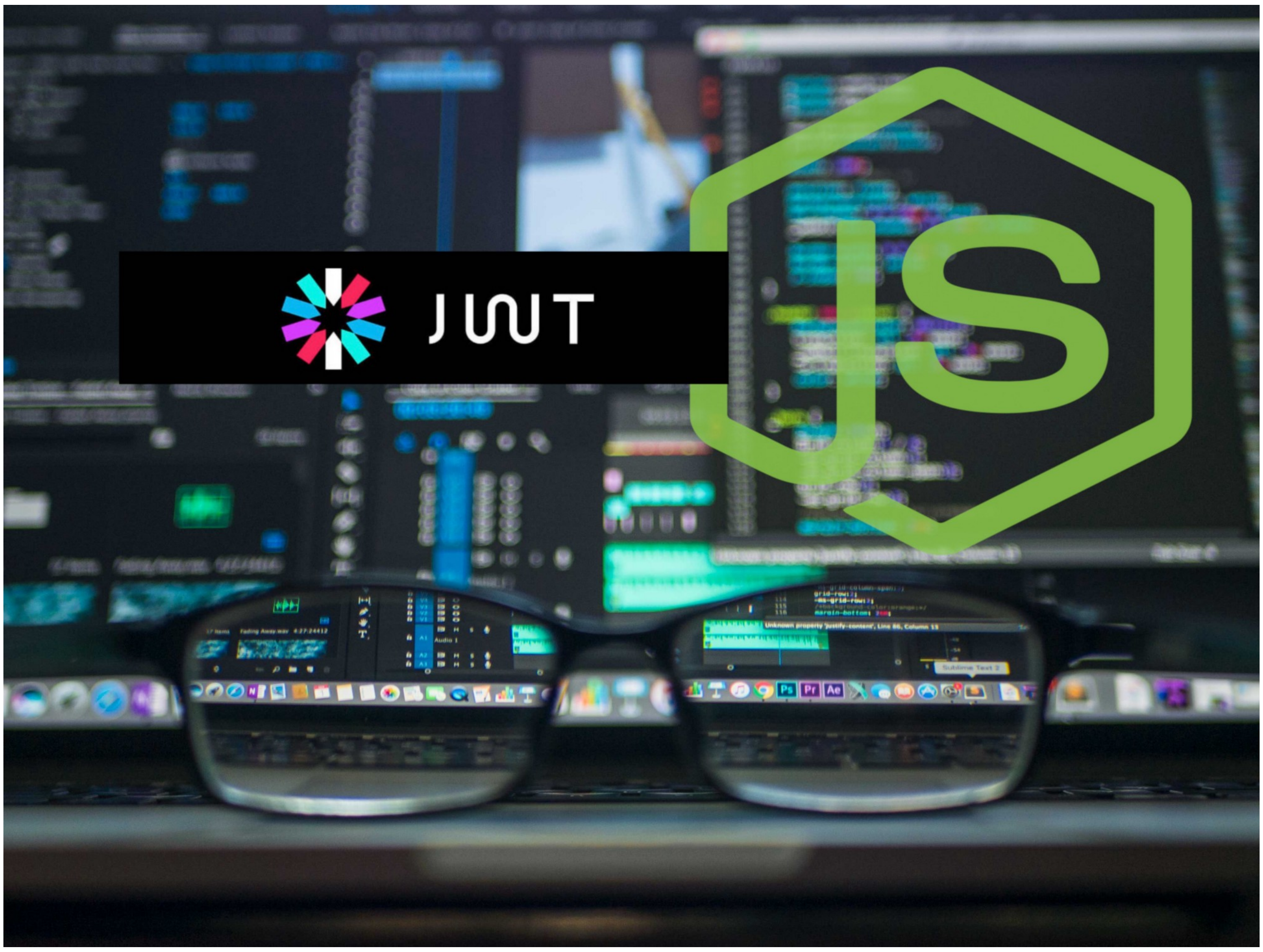
A guide for adding JWT token-based authentication to your single page Node.js applications

Naren Yellavula Follow
Aug 22, 2018 · 6 min read



Original Photo by Kevin Ku

JWT authentication is becoming very popular these days. The traditional authentication uses cookies and sessions. With the advent of Single Page Applications(SPA) and microservices, there is a need to look beyond the sessions. Any token based authentication serves that purpose. JWT is a type of token-based authentication. For every single request from a client to the server, a token is passed for authentication. It supports the stateless API calls.

In my previous article,

Developing Maintainable Websites with Single Page Applications
See why the state based User Interface makes sense in modern web development
medium.com

I d h
th ne
practical side of adding token based authentication especially JWT to SPA. The micro-services based architecture demands JWT because it is an open standard now.

> *"JSON Web Tokens are an open, industry standard* ***RFC 7519*** *method for representing claims securely between two parties."*

By the way, I wrote a cool book called "Building RESTful web services with Go" which has soup and tricks for handling web development problems.

https://www.amazon.com/Building-RESTful-Web-services-Go/dp/1788294289

JSON Web Tokens (JWT)

JSON web tokens are text strings that can be used by client and server to authenticate and share information easily. If you remember the necessary authentication, we do write information to the client by writing the cookie as a session variable. However, in JWT, a token is encoded from a data payload using a secret. That token is passed to the client. Whenever the client sends that token along with a request, the server validates it and sends back the response. See this diagram for a clear picture.

We can summarise above picture into following points:

- A client sends username/password combination to the server
- The server validates the authentication
- If authentication is successful, the server creates a JWT token else establishes an error response
- On successful authentication, the client gets JWT token in the response body
- Client stores that token in local storage or session storage.
- From next time, the client for making any request supplies the JWT token in request headers like this. `Authorization: Bearer <jwt_token>`
- Server upon receiving the JWT validates it and sends the successful response else error.

Yo s
ge u

get a whole lot of details here. https://jwt.io/introduction/

After reading that whole algorithm to generate tokens, you might be tempted to write your functionality. Hold on! There are many libraries available in different languages developed already. You plug them, and you get two main functionalities to generating JWT, validate JWT, etc.

Here I pick an NPM library called ***jsonwebtoken*** which is suggested by the JWT website rather than re-inventing the wheel.

Developing the sample application

Let us start with the ***package.json***. We need the following libraries in our app.

- Express JS — For serving requests
- jsonwebtoken — For writing and verifying JWT tokens

The ESLint packages are installed too because my editor use JS code linting.

Now do

```
npm install
```

We need to create a few more files for our project

```
touch server.js middleware.js config.js
```

server.js holds the main application logic. In middleware.js, we are going to define the JWT validation logic, and config.js is for storing secret for JWT hashing.

The tree structure looks like this.

```
.
├── config.js
├── middleware.js
├── package-lock.json
├── package.json
└── server.js
└── node_modules
```

Now let us add our config code in config.js

This secret will be read by JWT library while creating and validating tokens. In production, we need to store this secret in environment variable instead of a file.
In the middleware.js, we can write a function that acts as middleware to get a token from a request and proceeds only when the token is validated.

In the above code, we are doing the following things:

1. Capture headers with names 'x-access-token' or 'Authorization.'

2 d

3. Using **jwt** package and **secret** string, validate the token.
4. If anything goes wrong, return an error immediately before passing control to next handler.
5. Export the middleware function for other modules to use.

At this moment, we haven't written any code to create a token. We will do that next.

This might look lengthy but let us discuss the most exciting pieces from it.

```
let token = jwt.sign({username: username},

            config.secret,

            { expiresIn: '24h' // expires in 24 hours

            }

            );

// return the JWT token for the future API calls

res.json({

        success: true,

        message: 'Authentication successful!',

        token: token

});
```

jwt.sign function takes the payload, secret and options as its arguments. The payload can be used to find out which user is the owner of the token. Options can have an expire time until which token is valid. The generated token will be a string.

We are then sending the generated token back to the client in the response body. The client should preserve this token for future requests.

Note: In the above example, we are mocking the username and password, but in reality, those details should be fetched from the database server. For explaining the main point, I refrained from adding more details to this article.

In the main function, I added the code for starting my application server, attaching routes to middleware and handlers.

```
app.post('/login', handlers.login);

app.get('/', middleware.checkToken, handlers.index);
```

The second statement is a chain of handlers, where control first goes to middleware and then to the handler. Middleware function verifies JWT as we discussed above. Apart from index(/), you can add that middleware in front of all your REST endpoints to secure them with JWT authentication. Ex:

```
app.post('/v1/anotherResource', middleware.checkToken,
handlers.anotherHandler);
```

Let us run this app and see what we got here. Open a terminal and run this.

Server:

```
node server.js // starts server on 8000 port
```

Now try to make a curl request for home page(/).

Client:

```
curl -X GET http://localhost:8000

{"success":false,"message":"Auth token is not supplied"}
```

So our application is preventing us from accessing index without supplying the token. Now let us exchange our credentials for the token.

```
curl --header "Content-Type: application/json" \
  --request POST \
  --data '{"password":"password", "username":"admin"}' \
  http://localhost:8000/login

{
   "success":true,
   "message":"Authentication successful!",
"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWl
uIiwiaWF0IjoxNTM0OTMzNTY2LCJleHAiOjE1MzUwMTk5NjZ9.3xOdoxpK8hb42ykjMIl
6rwLafB63Y-EQNOO9fFamp68"
}
```

We can remake our last home page request by adding a bearer token.

```
curl -X GET \
  -H 'Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwiaWF0
IjoxNTM0OTI1MTQwLCJleHAiOjE1MzUwMTE1NDB9.MIcWFBzAr5WVhbaSa1kd1_hmEZse
po8fXqotqvAerKI' \
  http://localhost:8000

{
    "success": true,
    "message": "Index page"
}
```

So we get a successful response. This flow is how JWT authentication works, and we successfully integrated it with our app. The complete source code for this sample application s available at this Github link. https://github.com/narenaryan/node-jwt-integ

Here the client is CURL which is receiving the JWT token. However, when the browser(SPA) needs to access the REST API securely, it should save the token in its local storage. Here is a good discussion thread about the patterns for preserving tokens for a Single Page App. https://stackoverflow.com/questions/44133536/is-it-safe-to-store-a-jwt-in-localstorage-with-reactjs

I hope this article can help you to integrate JWT into your NodeJS REST API. If you have any queries, please contact me at @Narenarya3.

W

Nodejs JavaScript Authentication Jwt Software Development