Authentication and Authorization with JWTs in Express.js

By Janith Kasun • 13 Comments

Introduction

In this article, we will be talking about how JSON Web Tokens works, what are the advantages of them, their structure, and how to use them to handle basic authentication and authorization in Express.

You do not have to have any previous experience with JSON Web Tokens since we will be talking about it from scratch.

For the implementation section, it would be preferred if you have the previous experience with Express, Javascript ES6, and REST Clients.

What are JSON Web Tokens?

JSON Web Tokens (JWT) have been introduced as a method of communicating between two parties securely. It was introduced with the RFC 7519 specification by the *Internet Engineering Task Force* (IETF).

Even though we can use JWT with any type of communication method, today JWT is very popular for handling authentication and authorization via HTTP.

First, you'll need to know a few characteristics of HTTP.

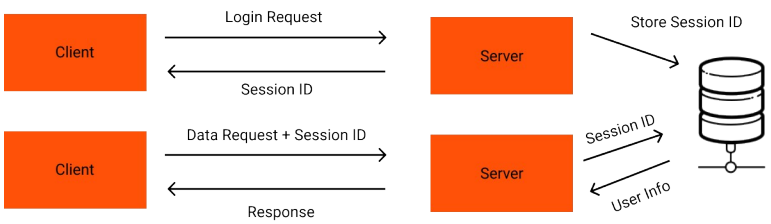
HTTP is a stateless protocol, which means that an HTTP request does not maintain state. The server does not know about any previous requests that were sent by the same client.

HTTP requests should be self-contained. They should include the information about previous requests that the user made in the request itself.

There are a few ways of doing this, however, the most popular way is to set a *session ID*, which is a reference to the user information.

The server will store this session ID in memory or in a database. The client will send each request with this sessions ID. The server can then fetch information about the client using this reference.

Here's is the diagram of how session-based authentication works:



Usually, this session ID is sent to the user as a cookie. We already discussed this in detailed in our

previous article Handling Authentication in Express.js.

On the other hand with JWT, when the client sends an authentication request to the server, it will send a JSON token back to the client, which includes all the information about the user with the response.

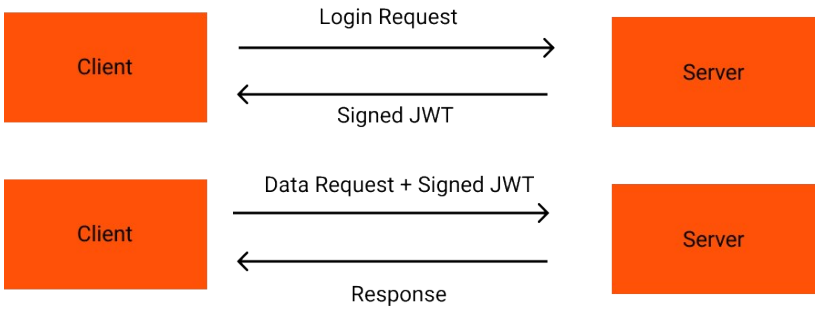
The client will send this token along with all the requests following that. So the server won't have to store any information about the session. But there is a problem with that approach. Anyone can send a fake request with a fake JSON token and pretend to be someone they are not.

For example, let's say that after authentication, the server sends back a JSON object with the username and the expiration time back to the client. So since the JSON object is readable, anyone can edit that information and send a request. The problem is, there is no way to validate such a request.

This is where the signing of the token comes in. So instead of just sending back a plain JSON token, the server will send a signed token, which can verify that the information is unchanged.

We will get into that in more detail later in this article.

Here is the diagram of how JWT works:



Structure of a JWT

Let's talk about the structure of a JWT through a sample token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHM
I6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

As you can see in the image, there are three sections of this JWT, each separated with a dot.

Sidebar: Base64 encoding is one way of making sure the data is uncorrupted as it does not compress or encrypt data, but simply encodes it in a way that most systems can understand. You can read any Base64 encoded text by simply decoding them.

The first section of the JWT is the header, which is a Base64-encoded string. If you decoded the header it would look something similar to this:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

The header section contains the hashing algorithm, which was used to generate the sign and the type of the token.

The second section is the payload that contains the JSON object that was sent back to the user. Since this is only Base64-encoded, it can easily be decoded by anyone.

It is recommended not to include any sensitive data in JWTs, such as passwords or personally identifiable information.

Usually, the JWT body will look something like this, though it's not necessarily enforced:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Most of the time, the `sub` property will contain the ID of the user, the property `iat`, which is shorthand for *issued at*, is the timestamp of when the token is issued.

You may also see some common properties such as `eat` or `exp`, which is the expiration time of the token.

The final section is the signature of the token. This is generated by hashing the string `base64UrlEncode(header) + "." + base64UrlEncode(payload) + secret` using the algorithm that is mentioned in the header section.

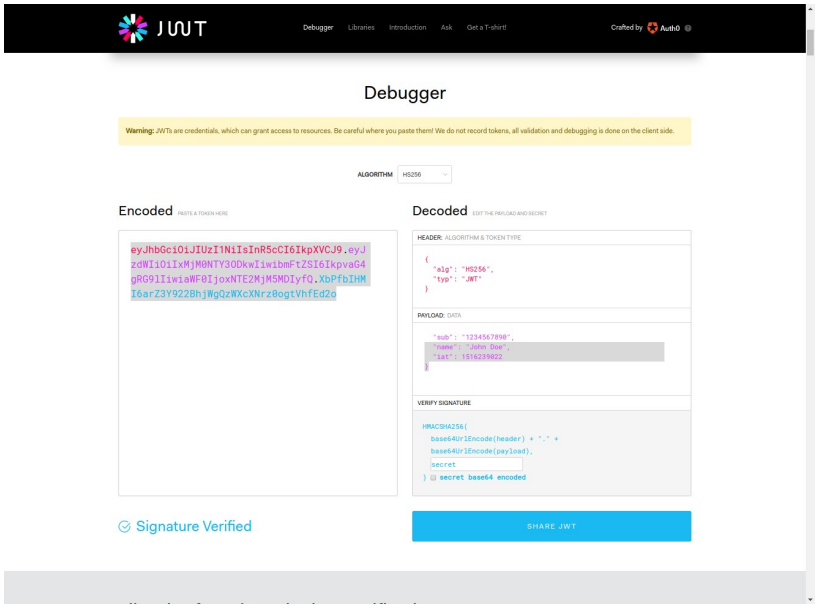
The `secret` is a random string which only the server should know. No hash can be converted back to the original text and even a small change of the original string will result in a different hash. So the `secret` cannot be reverse-engineered.

When this signature sends back to the server it can verify that the client has not changed any details in the object.

According to the standards, the client should send this token to the server via the HTTP request in a header called `Authorization` with the form `Bearer [JWT_TOKEN]`. So the value of the `Authorization` header will look something like:

```
Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MD
IyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

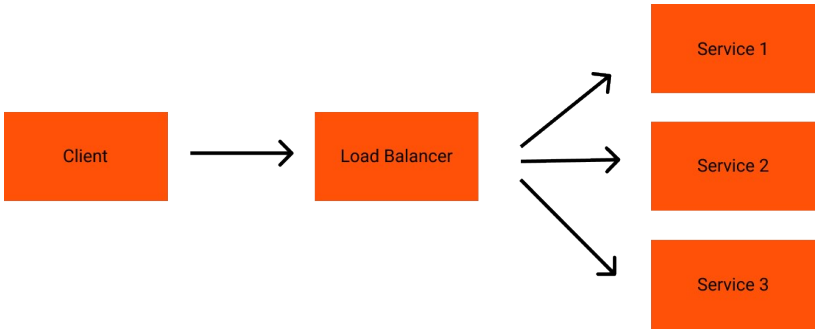
If you'd like to read more about the structure of a JWT token, you can check out our in-depth article, Understanding JSON Web Tokens. You can also visit jwt.io and play around with their debugger:



Advantage of Using JWT over Traditional Methods

As we have discussed earlier, JWT can contain all of the information about the user itself, unlike the session-based authentication.

This is very useful for scaling web apps, such as a web app with micro-services. Today, the architecture of a modern web app looks like something similar to this:



All of these services could be the same service, which will be redirected by the load balancer according to the resource usage (CPU or Memory Usage) of each server, or some different services such as authentication, etc.

If we use traditional authorization methods, such as cookies, we will have to share a database, like Redis, to share the complex information between servers or internal services. But if we share the secret across the micro-services, we can just use JWT and then no other external resources are

needed to authorize users.

Using JWT with Express

In this tutorial, we will be creating a simple micro-service-based web app to manage books in a library with two services. One service will be responsible for user authentication and the other will be responsible for managing books.

There will be two types of users - *administrators* and the *members*. Administrators will be able to view and add new books, whereas members will only be able to view them. Ideally they might also be able to edit or delete books. But to keep this article as simple as possible, we won't be going in to that much detail.

To get started, in your terminal initialize an empty Node.js project with default settings:

```
$ npm init -y
```

Then, let's install the Express framework:

```
$ npm install --save express
```

Authentication Service

Then, let's create a file called `auth.js`, which will be our authentication service:

```
const express = require('express');
const app = express();

app.listen(3000, () => {
    console.log('Authentication service started on port 3000');
});
```

Ideally, we should use a database to store user information. But to keep it simple let's create an array of users, which we will be using to authenticate them.

For every user, there will be the role - `admin` or `member` attached to their user object. Also, remember to hash the password if you are in a production environment:

```
const users = [
    {
        username: 'john',
        password: 'password123admin',
        role: 'admin'
    }, {
        username: 'anna',
        password: 'password123member',
        role: 'member'
    }
];
```

Now we can create a request handler for user login. Let's install the jsonwebtoken module, which is used to generate and verify JWT tokens.

Also, let's install the `body-parser` middleware to parse the JSON body from the HTTP request:

```
$ npm i --save body-parser jsonwebtoken
```

Now, let's these modules and configure them in the Express app:


```
const jwt = require('jsonwebtoken');
const bodyParser = require('body-parser');

app.use(bodyParser.json());
```

Now we can create a request handler to handle the user login request:

```
const accessTokenSecret = 'youraccesstokensecret';
```

This is your secret to sign the JWT token. You should never share this secret, otherwise a bad actor could use it to forge JWT tokens to gain unauthorized access to your service. The more complex this access token is, the more secure your application will be. So try to use a complex random string for this token:

```
app.post('/login', (req, res) => {
```

```
    // Read username and password from request body
    const { username, password } = req.body;

    // Filter user from the users array by username and password
    const user = users.find(u => { return u.username === username && u.password === password });

    if (user) {
        // Generate an access token
        const accessToken = jwt.sign({ username: user.username,  role: user.role }, accessTokenSecret);

        res.json({
            accessToken
        });
    } else {
        res.send('Username or password incorrect');
    }
});
```

In this handler, we have searched for a user that matches the username and the password in the request body. Then we have generated an access token with a JSON object with the username and the role of the user.

Our authentication service is ready. Let's boot it up by running:

```
$ node auth.js
```

After the authentication service is up and running, let's send a POST request and see if it works.

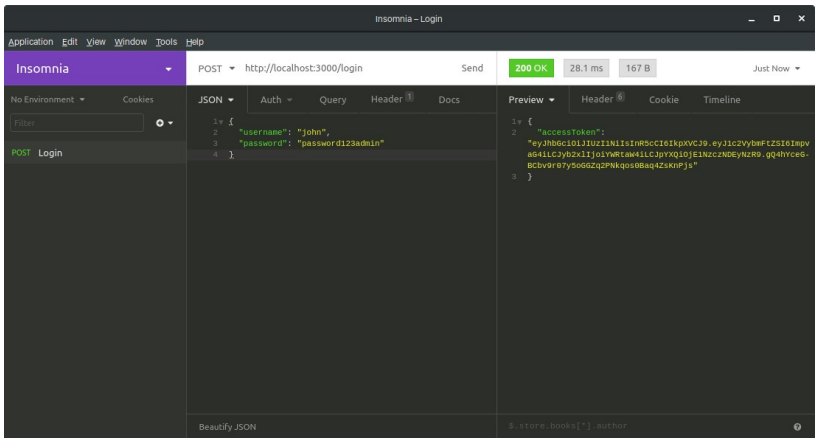
I will be using the rest-client Insomnia to do this. Feel free to use any rest-client you prefer or something like Postman to do this.

Let's send a post request to the `http://localhost:3000/login` endpoint with the following JSON:

```
{
    "username": "john",
    "password": "password123admin"
}
```

You should get the access token as the response:

```
{
  "accessToken": "eyJhbGciOiJIUz..."
}
```



Books Service

With that done, let's create a `books.js` file for our books service.

We'll start off the file by importing the required libraries and setting up the Express app:

```
const express = require('express');
const bodyParser = require('body-parser');
const jwt = require('jsonwebtoken');

const app = express();

app.use(bodyParser.json());

app.listen(4000, () => {
    console.log('Books service started on port 4000');
});
```

After the configuration, to simulate a database, let's just create an array of books:

```
const books = [
    {
        "author": "Chinua Achebe",
        "country": "Nigeria",
        "language": "English",
        "pages": 209,
        "title": "Things Fall Apart",
        "year": 1958
    },
    {
        "author": "Hans Christian Andersen",
        "country": "Denmark",
        "language": "Danish",
        "pages": 784,
        "title": "Fairy tales",
```

```
        "year": 1836
    },
    {
        "author": "Dante Alighieri",
        "country": "Italy",
        "language": "Italian",
        "pages": 928,
        "title": "The Divine Comedy",
        "year": 1315
    },
];
```

Now, we can create a very simple request handler to retrieve all books from the database:

```
app.get('/books', (req, res) => {
    res.json(books);
});
```

Because our books should be only visible to authenticated users. We have to create a middleware for authentication.

Before that, create the access token secret for the JWT signing, just like before:

```
const accessTokenSecret = 'youraccesstokensecret';
```

This token should be the same one used in the authentication service. Due to the fact the secret is shared between them, we can authenticate using the authentication service and then authorize the users in the book service.

At this point, let's create the Express middleware that handles the authentication process:

```
const authenticateJWT = (req, res, next) => {
    const authHeader = req.headers.authorization;

    if (authHeader) {
        const token = authHeader.split(' ')[1];

        jwt.verify(token, accessTokenSecret, (err, user) => {
            if (err) {
                return res.sendStatus(403);
            }

            req.user = user;
            next();
        });
    } else {
        res.sendStatus(401);
    }
};
```

In this middleware, we read the value of the authorization header. Since the `authorization` header has a value in the format of `Bearer [JWT_TOKEN]`, we have split the value by the space and separated the token.

Then we have verified the token with JWT. Once verified, we attach the `user` object into the request and continue. Otherwise, we will send an error to the client.

We can configure this middleware in our GET request handler, like this:

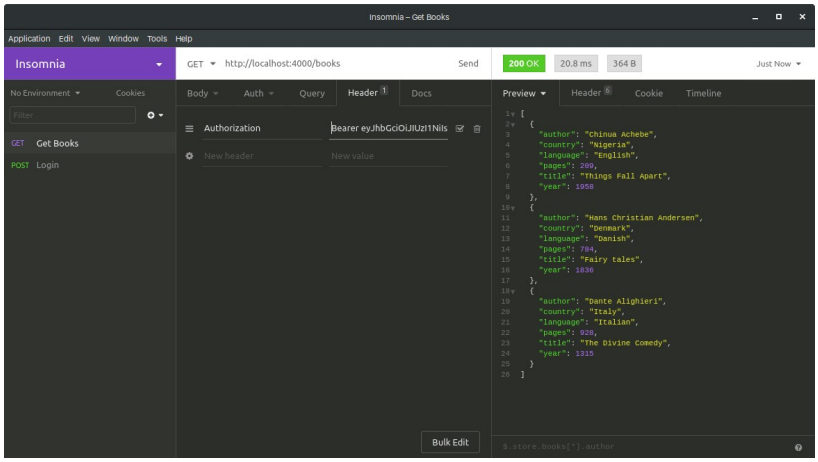
```
app.get('/books', authenticateJWT, (req, res) => {
    res.json(books);
});
```

Let's boot up the server and test if everything's working correctly:

```
$ node books.js
```

Now we can send a request to the `http://localhost:4000/books` endpoint to retrieve all the books from the database.

Make sure you change the "Authorization" header to contain the value "Bearer [JWT_TOKEN]", as shown in the image below:



Finally, we can create our request handler to create a book. Because only an `admin` can add a new book, in this handler we have to check the user role as well.

We can use the authentication middleware that we have used above in this as well:

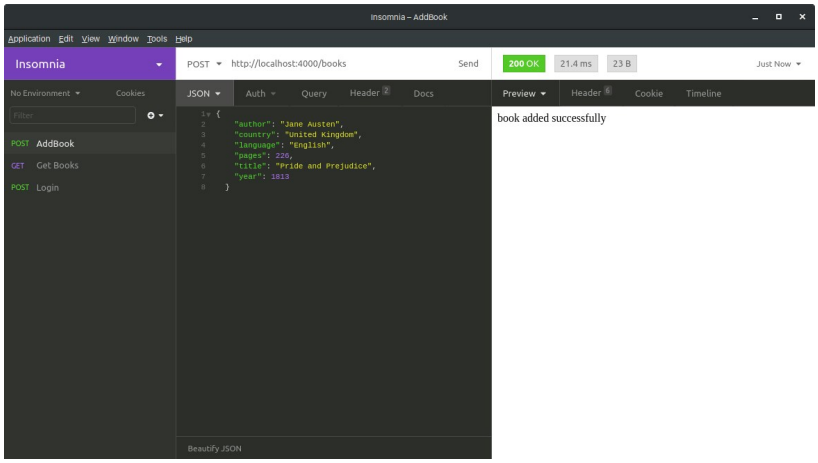
```
app.post('/books', authenticateJWT, (req, res) => {
    const { role } = req.user;

    if (role !== 'admin') {
        return res.sendStatus(403);
    }


    const book = req.body;
    books.push(book);

    res.send('Book added successfully');
});
```

Since the authentication middleware binds the user to the request, we can fetch the `role` from the `req.user` object and simply check if the user is an `admin`. If so, the book is added, otherwise, an error is thrown.

Let's try this with our REST client. Log in as an `admin` user (using the same method as above) and then copy the `accessToken` and send it with the `Authorization` header as we have done in the previous example.

Then we can send a POST request to the `http://localhost:4000/books` endpoint:

```
{
    "author": "Jane Austen",
    "country": "United Kingdom",
    "language": "English",
    "pages": 226,
    "title": "Pride and Prejudice",
    "year": 1813
}
```



Token Refresh

At this point, our application handles both authentication and authorization for the book service, although there's a *major* flaw with the design - the JWT token never expires.

If this token is stolen, then they will have access to the account forever and the actual user won't be able to revoke access.

To remove this possibility, let's update our login request handler to make the token expire after a specific period. We can do this by passing the `expiresIn` property as an option to sign the JWT.

When we expire a token, we should also have a strategy to generate a new one, on the event of an expiration. To do that, we'll create a separate JWT token, called a *refresh token*, which can be used to generate a new one.

First, create a refresh token secret and an empty array to store refresh tokens:

```
const refreshTokenSecret = 'yourrefreshtokensecrethere';
const refreshTokens = [];
```

When a user logs in, instead of generating a single token, generate both refresh and authentication tokens:

```
app.post('/login', (req, res) => {
    // read username and password from request body
    const { username, password } = req.body;

    // filter user from the users array by username and password
    const user = users.find(u => { return u.username === username && u.password === password });
```

```
    if (user) {
        // generate an access token
        const accessToken = jwt.sign({ username: user.username, role: user.role }, accessTokenSecret, { expiresIn:
 '20m' });
        const refreshToken = jwt.sign({ username: user.username, role: user.role }, refreshTokenSecret);

        refreshTokens.push(refreshToken);

        res.json({
            accessToken,
            refreshToken
        });
    } else {
        res.send('Username or password incorrect');
    }
});
```

And now, let's create a request handler that generated new tokens based on the refresh tokens:

```
app.post('/token', (req, res) => {
    const { token } = req.body;

    if (!token) {
        return res.sendStatus(401);
    }

    if (!refreshTokens.includes(token)) {
        return res.sendStatus(403);
    }

    jwt.verify(token, refreshTokenSecret, (err, user) => {
        if (err) {
            return res.sendStatus(403);
        }

        const accessToken = jwt.sign({ username: user.username, role: user.role }, accessTokenSecret, { expiresIn:
 '20m' });

        res.json({
            accessToken
        });
    });
});
```

But there is a problem with this too. If the refresh token is stolen from the user, someone can use it to generate as many new tokens as they'd like.

To avoid this, let's implement a simple `logout` function:

```
app.post('/logout', (req, res) => {
    const { token } = req.body;
    refreshTokens = refreshTokens.filter(token => t !== token);

    res.send("Logout successful");
});
```

When the user requests to logout, we will remove the refresh token from our array. It makes sure that when the user is logged out, no one will be able to use the refresh token to generate a new authentication token.

Conclusion

In this article, we have introduced you to JWT and how to implement JWT with Express. I hope that now you have a piece of good knowledge about how JWT works and how to implement it in your project.

As always the source code is available in GitHub.

node, javascript, express, jwt

About Janith Kasun

Colombo, Sri Lanka Twitter

‹ Previous Post

Next Post ›

Recent Posts

Matplotlib Stack Plot - Tutorial and Examples

JavaScript: Get Number of Days Between Dates

How to Split an Array Into Even Chunks in JavaScript

Tags

ai algorithms amqp angular announcements apache apache commons api arduino artificial intelligence

Follow Us

Twitter Facebook RSS