

Open in app ↗



JWT Authentication in Nodejs — Refresh JWT with Cookie-based Token

Suneel Kumar · [Follow](#)

6 min read · Feb 3, 2023



Listen



Share



More

Photo by [Ferhat Deniz Fors](#) on [Unsplash](#)

JSON Web Tokens (JWTs) are a popular method of authentication that allow you to securely transmit information between parties as a JSON object. In this article, we'll be diving into the details of JWT authentication in a Node.js application and exploring the use of refresh tokens to extend the life of our JWTs.

To start, let's take a quick overview of JWT and its components.

What is JWT?

A JWT is a compact and self-contained JSON object that contains information about the authentication of a user. This information can be verified and trusted because it is digitally signed using a secret key. The main components of a JWT are:

1. **Header:** This contains information about how the JWT is encoded, such as the algorithm used for signing the token.
2. **Payload:** This contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.
3. **Signature:** This is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way. The signature is created by taking the encoded header, the encoded payload, and a secret key, then signing them using a specified algorithm.

Here's an example of a JWT in its compact form:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaC
```

Setting up the environment

Before we dive into the implementation, let's set up our development environment. We'll be using the following tools and packages:

1. **Node.js:** We'll be using Node.js as our server-side language.
2. **Express:** This is a popular web framework for Node.js that we'll be using to handle HTTP requests.
3. **jsonwebtoken:** This is a popular library for generating and verifying JWTs in Node.js.

Let's start by creating a new Node.js project using the following command:

```
npm init
```

Next, let's install the required packages by running the following command:

```
npm install express jsonwebtoken
```

Implementing JWT Authentication

Now that we've set up our environment, let's start implementing JWT authentication in our Node.js application.

First, let's create a new file called `index.js` and add the following code to set up our Express application:

```
const express = require('express');
const app = express();
const port = 3000;

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}`);
});
```

Next, let's create a simple route that will return a message if the user is authenticated:

```
app.get('/protected', (req, res) => {
  res.send('Welcome to the protected route');
});
```

Now, let's implement the authentication process by adding middleware that will check for the presence of a valid JWT in the `Authorization` header of incoming requests:

```
const jwt = require('jsonwebtoken');
const secretKey = 'secret_key';

const authenticate = (req, res, next) => {
```

```
const token = req.headers['authorization'];
if (!token) {
  return res.status(401).send('Access Denied. No token provided.');
```



```
  }

  try {
    const decoded = jwt.verify(token, secretKey);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(400).send('Invalid Token.');
```



```
  }
};

app.get('/protected', authenticate, (req, res) => {
  res.send('Welcome to the protected route');
});
```

In the code above, we imported the `jsonwebtoken` library and created a secret key for signing and verifying our JWTs. Then, we created a middleware function called `authenticate` that will be used to check for a valid JWT in the `Authorization` header of incoming requests.

If a token is present, we use the `verify` method from the `jsonwebtoken` library to decode the token and check its validity. If the token is valid, we add the decoded information to the `req` object and call the `next` function to proceed to the next middleware or route handler.

If no token is provided or if the token is invalid, we return an error message with a `401` or `400` status code, respectively.

Generating JWTs

Now that we have our authentication process in place, let's create a route for generating JWTs. This could be done when a user logs in, for example.

```
app.post('/login', (req, res) => {
  const user = {
    id: 1,
    username: 'john.doe'
  };

  const token = jwt.sign({ user }, secretKey, { expiresIn: '1h' });
```

```
res.header('Authorization', token).send(user);  
});
```

In the code above, we created a `/login` route that will generate a JWT when hit with a `POST` request. We created a mock user object and used the `sign` method from the `jsonwebtoken` library to generate a JWT. We specified the secret key and set the `expiresIn` option to `1h`, which means that the token will expire after one hour.

Finally, we added the generated JWT to the `Authorization` header of the response and sent the user information back to the client.

Implementing Refresh Tokens

One issue with JWTs is that they are short-lived and need to be refreshed regularly. To address this, we can use refresh tokens.

Refresh tokens are separate tokens from access tokens, and they can be used to generate new access tokens. This allows us to keep our access tokens short-lived for security reasons while still allowing the user to remain authenticated for longer periods of time.

In this section, we'll explore the implementation of refresh tokens using cookies.

Let's start by modifying the `/login` route to generate both an access token and a refresh token:

```
app.post('/login', (req, res) => {  
  const user = {  
    id: 1,  
    username: 'john.doe'  
  };  
  
  const accessToken = jwt.sign({ user }, secretKey, { expiresIn: '1h' });  
  const refreshToken = jwt.sign({ user }, secretKey, { expiresIn: '1d' });  
  
  res  
    .cookie('refreshToken', refreshToken, { httpOnly: true, sameSite: 'strict' })  
    .header('Authorization', accessToken)  
    .send(user);  
});
```

Next, we'll create a new route for refreshing access tokens. This route will receive a refresh token from the client and use it to generate a new access token:

```
app.post('/refresh', (req, res) => {  
  const refreshToken = req.cookies['refreshToken'];  
  if (!refreshToken) {  
    return res.status(401).send('Access Denied. No refresh token provided.');  }  
  
  try {  
    const decoded = jwt.verify(refreshToken, secretKey);  
    const accessToken = jwt.sign({ user: decoded.user }, secretKey, { expiresIn: 60 * 60 * 24 });  
  
    res  
      .header('Authorization', accessToken)  
      .send(decoded.user);  
  } catch (error) {  
    return res.status(400).send('Invalid refresh token.');  }  
});
```

In the code above, we created a new route for refreshing access tokens. This route checks for the presence of a refresh token in the cookies, and if a token is found, we use the `verify` method to check its validity. If the refresh token is valid, we generate a new access token with the same information as the original token and send it back to the client.

Finally, we can update the `authenticate` middleware to check for both the access token and refresh token and refresh the access token if necessary:

```
const authenticate = (req, res, next) => {  
  const accessToken = req.headers['authorization'];  
  const refreshToken = req.cookies['refreshToken'];  
  
  if (!accessToken && !refreshToken) {  
    return res.status(401).send('Access Denied. No token provided.');  }  
  
  try {  
    const decoded = jwt.verify(accessToken, secretKey);  
    req.user = decoded.user;  
    next();  
  } catch (error) {  
    return res.status(400).send('Invalid access token.');  }  
};
```

```
    } catch (error) {
      if (!refreshToken) {
        return res.status(401).send('Access Denied. No refresh token provided.');
```

```
      }

      try {
        const decoded = jwt.verify(refreshToken, secretKey);
        const accessToken = jwt.sign({ user: decoded.user }, secretKey, { expires
```

```
      res
        .cookie('refreshToken', refreshToken, { httpOnly: true, sameSite: 'stri
```

```
        .header('Authorization', accessToken)
        .send(decoded.user);
      } catch (error) {
        return res.status(400).send('Invalid Token.');
```

```
      }
    }
  }
};
```

In the updated code, we first check for the presence of both access and refresh tokens. If only an access token is provided, we check its validity and continue the request if it's valid. If the access token is invalid, we check for the presence of a refresh token. If a refresh token is found, we verify its validity and generate a new access token based on the information in the refresh token. If the refresh token is invalid, we send an error message to the client.

With these changes in place, we now have a complete JWT authentication system with a refresh token mechanism. To use this system, clients can first make a request to the `/login` route to receive an access token and refresh token. They can then include the access token in the `Authorization` header of all subsequent requests to access protected routes. If the access token expires, the client can make a request to the `/refresh` route to receive a new access token.

We hope this article has been helpful in demonstrating how to implement JWT authentication with a refresh token mechanism in Node.js. This is just one of the many ways to implement JWT authentication and there are many other ways to improve this system based on the specific needs of your application.

Jwt

Nodejs

JavaScript

Authentication

Programming

[Follow](#)

Written by Suneel Kumar

1.4K Followers

Freelancing, Blogging. Explore with me! Need help in creating blog/website?>>>please visit <https://www.linkedin.com/in/suneel-kumar-nodejs/>

More from Suneel Kumar



Suneel Kumar

Design Patterns in Node.js

Design patterns are proven solutions to common programming problems. They provide tested architectures and interfaces that can make your...

7 min read · Aug 15, 2023




595



4







 Suneel Kumar

Implementing Role-Based Access Control (RBAC) in Node.js

Role-Based Access Control (RBAC) is a crucial aspect of application security. It provides a structured approach to manage and restrict...

9 min read · Oct 19, 2023

 355  3



 Suneel Kumar

Message Queue in Node.js with BullMQ and Redis

A message queue is a mechanism for communicating between different components of an application in an asynchronous way. It allows...

9 min read · Sep 27, 2023



60



2



```
ReactDOM.render((  
  <BrowserRouter>  
    <Switch>  
      <Route path="/login" component={Login} />  
      <ProtectedRoute exact={true} path="/" component={Dashboard} />  
      <ProtectedRoute path="/settings" component={Settings} />  
      <ProtectedRoute component={Dashboard} />  
    </Switch>  
  </BrowserRouter>  
, document.getElementById('root'));
```



Suneel Kumar

How I Have Mastered Closures in React 🚀

React, a powerful and widely-used JavaScript library for building user interfaces, offers developers a flexible and efficient way to create...

3 min read · Dec 31, 2023



155



4



See all from Suneel Kumar

Recommended from Medium



Dev Balaji

JWT Authentication in Node.js: A Practical Guide

Implementing authentication for a web application using Node.js typically involves several steps. “Token-based authentication” is a common...

3 min read · Sep 7, 2023



161



1





Saniaalikhan in Stackademic

Mastering Security: Role-Based Access Control in Node.js with JWT

In the dynamic landscape of web applications, ensuring secure and role-based access is paramount. Today, we embark on a journey to fortify...

3 min read · Oct 12, 2023



41



1

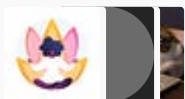


Lists



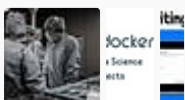
General Coding Knowledge

20 stories · 875 saves



Stories to Help You Grow as a Software Developer

19 stories · 777 saves



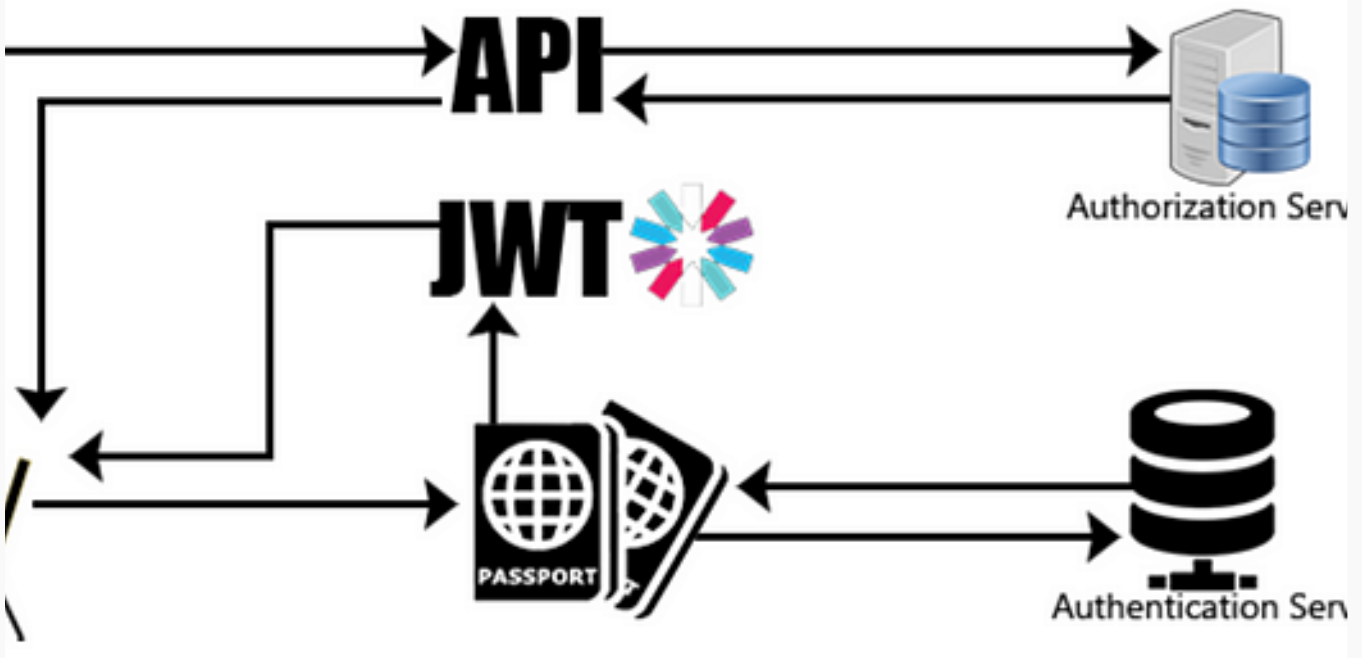
Coding & Development


11 stories · 424 saves



ChatGPT

21 stories · 438 saves



 Pawan Kumar

Authentication and Authorization in Node.js: A Comprehensive Guide

Authentication and authorization are two fundamental concepts in web application security. They ensure that users have the right level of...

3 min read · Oct 4, 2023

 53 



 Suneel Kumar

Implementing Role-Based Access Control (RBAC) in Node.js

Role-Based Access Control (RBAC) is a crucial aspect of application security. It provides a structured approach to manage and restrict...

9 min read · Oct 19, 2023



355



3



Mangesh Pal

Node.js JWT Authentication With HTTP Only Cookie (by mangesh Pal)

3 min read · Sep 5, 2023



Understanding Role-Based Access Control



CISO



Manager



Basic User



Ananya Sharma

Role-Based Access Control in Nodejs

Introduction

8 min read · Aug 17, 2023



285



2



See more recommendations