Task

Authentication With JWT

Visit our website

Introduction

WELCOME TO THE AUTHENTICATION WITH JWT TASK!

Up until now, you've been exposed to web development languages, frameworks, libraries, and paradigms. One crucial (and final) component to web development that you haven't yet made contact with is authentication. That is, how to ensure your users only have access to the facets that they're supposed to. In this task, we'll look at one popular way of doing so.

REST AND EARLY AUTHENTICATION

In the early days of the internet, RESTful requests were designed to be stateless, meaning each request had to contain all necessary information without relying on prior ones. For example, to buy a product, a single request needed to authenticate the user, select products, and process payment.

Though this led to longer requests, it reduced the number of requests required, which was important given the slow, unreliable Internet and limited hardware capabilities at the time.

The first web authentication method, basic authentication, included the username and password in every request header. It was simple but risky if used without HTTPS, as the credentials could be intercepted by methods such as **man-in-the-middle attacks**. While basic authentication is still used for simple API requests, it's always over HTTPS and typically not in browsers.

CONTEMPORARY AUTHENTICATION

Today, technology has improved exponentially to where you can make ten requests, with reasonable confidence, and expect them all to complete before a second has passed – even on basic hardware and a simple Internet connection. Through multiple iterations, many different authentication techniques came about. Most of them that are still used today work something like this:

1. The client sends the username and password to an authentication (auth) endpoint.
2. The auth endpoint checks the data and, if legit, generates an auth token, which is relevant to the requesting user's session.
3. The client stores the token and adds it to the header of further requests.
4. The server checks the token every time it receives a request and uses it to determine which user is making the request.

There are many different ways to implement this process. Some have multiple-step authentication, and some require extra internal authentication steps with regular requests. All of them expend a great deal of energy to ensure this generated token is secure and hacker-proof.

WHAT ARE JSON WEB TOKENS?

JSON web tokens (JWTs) are a URL-safe method of transferring information between two parties. JWTs are mainly used to transfer information securely between two endpoints: server and client (web browser). Their purpose is to authenticate and authorise processes and often help in the stateless exchange of information.

COMPONENTS OF A JWT

The components of a JWT consist of mainly three things:

- Header
- Payload
- Signature

Let's look at all three of these constituents of JWTs:

Headers

Headers provide information on how a JWT is encoded, and consists of cryptic algorithms that help in securing it. It's a JSON object with two main properties:

- **alg:** Short for "algorithm", this property specifies the algorithm used to sign the JWT. Some of the most popular algorithms are **HMAC**-**SHA256**, RSA, and ESDSA. The choice of algorithm alters the security and integrity of the token.
- **typ:** This property indicates the type of token. For JWT, this is specifically set to JWT.

An example of a JWT header is given below:

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

Base64 encoding is a way to represent binary data in a textual format that is safe to be transmitted over various protocols. When a Base64 header is encoded, it is converted from binary data to ASCII characters. To encode any header content into Base64 encoding, you can use any programming language that supports the use of Base64 libraries/modules. The original header above is converted into a Base64 encoded header using the **JWT platform**.

The Base64 encoded header used for a JWT is given below:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Payload

The payload of a JWT consists of actual claims or statements about the subject (user) or device, and it also contains additional data. Claims are assertions made about the actual subject or device and have three types:

- **Reserved claims:** These are pre-defined claims that have several special meanings, such as "iss", which stands for issuer; "sub", which means subject; "exp", which stands for expiration; and "iat", which means issues at time, etc. These claims convey standard information about the token.
- **Public claims:** These are customised claims made by the parties involved in the token exchange. They also provide extra information regarding a specific use case.
- **Private claims:** These claims are used to share private information between parties in a private manner. These claims are not registered in any official claims namespaces.

An example of a payload is given below:

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

The Base64 encoded payload is given below:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
```

To produce the signature, we use the algorithm in the header, HMAC-SHA256. It's a hashing function that encodes a message one-way into a consistent hash. Our HS256 algorithm adds a secret key to the process. You need the header, payload, and secret key to create the signature, which will always produce the same hash when given the same inputs.

Signature

Creating a signature for a token entails taking a header, a payload, and a secret key and applying a specific algorithm to it. The key can be private or public, depending on the algorithm used for the payloads. The signatures ensure that the tokens are authentic and their integrity is maintained. Only the parties who know the key can generate a valid signature if the key is private. The process of creating a signature is mentioned below:

1. Base64 URL encoded header.
2. Base64 URL encoded payload.
3. Concatenating the header and the payload with a full stop.
4. Applying a specific algorithm to the concatenated string and secret key.

An example of the resulting signature is given below:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret key
)

header = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9'
payload = 'eyJpZCI6MTIzNCwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9'
msg = header + '.' + payload
sig = HS256('secret-key', msg).digestBase64()
```

Combining all three of these components creates a token, which ensures that information is passed securely between two parties. An example of a signature after encoding is given below:

```
nENNz0tGIfhz2ehg4XBoe30K4nLg2vPs8JQjfKde_m8
```

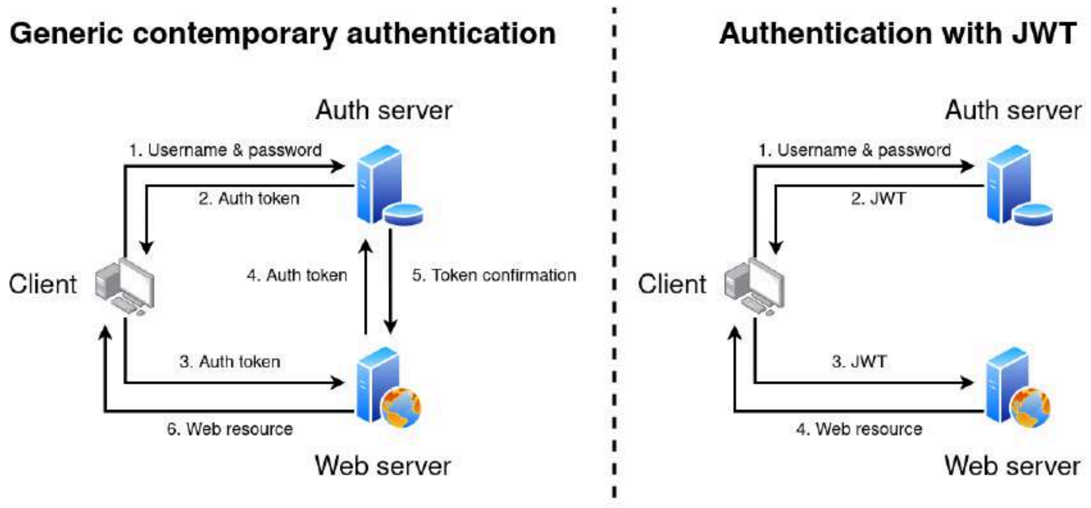
The final JWT looks something like the following:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp
vaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.nENNz0tGIfhz2ehg4XBoe30K4nLg2vPs8JQjfKde
_m8
```

JWTs contain user information like ID, name, and admin level. Their payload can be decoded with Base64, making it easy for anyone with the token to read the data. Thus, sensitive information, such as passwords or credit card numbers, should not be included. Instead, JWTs are designed to prevent tampering through a signature created with a secret key, ensuring the payload remains unchanged.

While traditional authentication methods have been used for years, JWTs offer a more efficient way to prevent tampering during transit and validate data using encryption keys. The main goal of authentication is to control user access; for instance, admins can delete posts while authors can only edit them. Traditional methods require the server to check the auth token against the database, which can be slow.

JWTs simplify this process by embedding permission data within the token itself, allowing the server to validate and grant or deny access without additional database lookups.



With JWTs, there is no direct communication between the web and auth servers. JWTs contain all the necessary authorisation data, and their legitimacy can be validated independently.

IMPLEMENTATION OF JWTS

To get started with implementing JWTs in an Express app, let's create a basic server that will receive user login credentials and respond accordingly. In this example, we'll use Express's built-in JSON body parsing, which allows us to easily handle data sent by the client.

Setting up the express server

Start by creating a new Express application and saving the main file as **index.js**. Then, add the following code to set up and run a basic server:

```
// Import Express
const express = require('express')
const app = express()
const PORT = 8000

// Enable JSON parsing for incoming request bodies
app.use(express.json())

// User login route
app.post('/login', (req, res) => {
  // Extracting username and password from the request body
  const username = req.body.username
  const password = req.body.password

  // For now, we'll just respond with the received username and password
  res.send(`Username: ${username}\nPassword: ${password}`)
})

// Start the server
app.listen(PORT, () => console.log(
  `Server is running at http://localhost:${PORT}`
))
```

In the code above, we set up an Express application with a `/login` endpoint to handle user login. We start by importing the Express library and initialising an Express app, then set it to listen on port **8 000** for incoming requests.

Using `app.use(express.json())`, we enable JSON parsing for request bodies, which allows the server to handle JSON data sent by clients and makes it accessible through `req.body`. This is particularly useful for APIs, as it ensures that any JSON data sent in requests can be accessed without extra middleware.

The `/login` route handles POST requests, where we retrieve the username and password from `req.body`, which is the data sent by the client. In this example, the server responds with a message showing the received credentials, demonstrating how data can be received and processed. Finally, we start the server, which outputs a console message confirming it's running and ready to accept requests.

Installing dependencies

Before running the Express server, navigate to the project directory (the same folder where you initialised the NPM package and where **package.json** is located). Then, install Express by entering the following command in your terminal:

```
npm install express
```

Running the application

To start the server, use the following command. If you choose to install and use nodemon for automatic restarts, your command may vary:

```
node index.js
```

By now, you should be familiar with using Postman. You can use Postman to create POST requests to `http://localhost:8000/login` with the following body:

```
{
 "username": "zama",
 "password": "abcdef"
}
```

In Postman, be sure to include the `Content-Type: application/json` header to specify that the body content is JSON. You can do this by selecting "JSON" in the

“Body” tab, which will automatically set the header for you. This is a standard practice for working with RESTful APIs.

If all goes well, your Express app server should respond with the following, confirming that it understands your POST request:

```
Username: zama
Password: abcdef
```

Now that we have a basic Express app running, let’s outline our intended design for a secure and organised API. We will implement the following three primary endpoints:

1. **Authentication endpoint (/login):** This endpoint will handle user authentication. It will validate the user’s credentials (such as the username and password) and respond with a JWT if the login was successful. The JWT will then allow the client to access protected resources by including it in future requests.
2. **Resource endpoint (/resource):** This endpoint will provide access to general resources for authenticated users. To access this endpoint, a user must include a valid JWT in the request, allowing the server to verify the user’s identify before granting access.
3. **Admin resource endpoint (/admin_resource):** This endpoint will provide access to resources specifically intended for admin users. In addition to requiring a valid JWT, this endpoint will also check for specific claims or permissions within the JWT to confirm the user has admin privileges before granting access.

Currently, we only have the `/login` endpoint set up, which simply accepts and displays the username and password. In later steps, we will expand upon this endpoint to include validation and JWT generation. We will also implement the `/resource` and `/admin_resource` endpoints to demonstrate how access control is enforced based on the JWT provided.

Creating the authentication endpoint

Let's begin by updating the `/login` authentication endpoint. Firstly, we will implement basic username and password validation. If the credentials are incorrect, we will respond with HTTP status code **403**, which is a standard response code for authentication failure in RESTful APIs.

To set up this basic validation, update the `/login` endpoint by removing the previous `res.send()` statement and replacing it with the following code:

```
if (username === 'zama' && password === 'secret') {
   // To-do: Generate a JWT here in the next step
} else {
   res.status(403).send({ error: 'Incorrect login!' })
}
```

In this example, the username and password are hardcoded for demonstration purposes. In a real application, however, you would validate these credentials against a secure database. Alternatively, as a simpler next step, you could check the credentials against an array of user objects. Note that we return the error message as a JSON object. While it could be a simple string, it's good practice to keep the request and response formats consistent throughout your application.

After adding this basic validation, restart your app if you're not using nodemon. Test the endpoint by entering incorrect credentials in your request body, and you should see the error message: `{"error": "Incorrect login!"}`.

Introducing JWTs

The next step is for us to use JWTs to secure access for authenticated users. The most popular library for working with JWTs in Node.js is **jsonwebtoken**. To install it, run the following command:

```
npm install jsonwebtoken
```

After installing jsonwebtoken, add the following to the top of your **index.js** to import the installed library:

```
const jwt = require('jsonwebtoken')
```

Now, we can generate a JWT for successfully authenticated users. Replace the `// To-do` comment in the `/login` endpoint with code that creates and sends a JWT:

```
if (username === 'zama' && password === 'secret') {
   // User payload for JWT
   const payload = {
      name: username,
      admin: false
   }

   // Generate the JWT
   const token = jwt.sign(payload, 'jwt-secret', { algorithm: 'HS256' })

   // Send the token in the response
   res.send({ token })
} else {
   res.status(403).send({ error: 'Incorrect login!' })
}
```

In the above code, we have used `jwt.sign()` to create a token based on the payload object that includes the user's information, such as `name` and `admin` status. We sign the token with the secret key (`'jwt-secret'`) and specify an encryption algorithm, `HS256`. The jsonwebtoken library helps to simplify JWT generation, allowing us to avoid the need to manually implement complex algorithms.

You can now restart your application if required. Try logging in with the correct credentials, and you should receive a response containing your JWT in the following format:

```
{
   "Token":
"eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiemFtYSIsImFkbWluIjpmYWxzZX0.KTo5xXD2ecZWEa
ABPny6Y2Z6dM0AxPcdtWAAW_TcKTM"
}
```

Implementing the /resource endpoint

To add the `/resource` endpoint, place the following code above the `app.listen()` line in your **index.js** file. This order ensures that all route handlers are defined before the server starts listening for requests.

```
// Route to access a general resource, requiring a valid JWT for
authentication
app.get('/resource', (req, res) => {
  // Extract the token from the authorisation header
  const authHeader = req.headers['authorization'];
  let token;

  // Check if the authorisation header includes the “Bearer” prefix
  if (authHeader && authHeader.startsWith('Bearer ')) {
    // If it does, split to extract the token part after “Bearer”
    token = authHeader.split(' ')[1];
  } else {
    // If no Bearer prefix, assume the entire header is the token
    token = authHeader;
  }

  // If no token is found, respond with a “401 Unauthorized” error
  if (!token) {
    return res.status(401).send({ error: 'Token missing!' });
  }

  try {
    // Verify the JWT using the secret key
    const decoded = jwt.verify(token, 'jwt-secret');

    // If verification succeeds, send a personalised message
    res.send({
      message: `Hello, ${decoded.name}! Your JSON Web Token has been
verified.`
    });
  } catch (err) {
    // If verification fails, respond with a “401 Unauthorized” error
    res.status(401).send({ error: 'Invalid JWT!' });
  }
});
```

JWTs are typically included within the authorisation header of requests in the format `Authorization: Bearer <token>`, following REST API conventions. This code first checks if the header starts with `Bearer`:

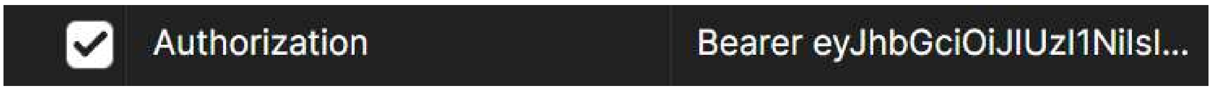
- **With the Bearer prefix:** If the header includes `Bearer`, `.split(' ')[1]` extracts only the token portion.
- **Without the Bearer prefix:** If the token is provided directly (e.g., `Authorization: <token>`), the code uses the entire header value as the token.

Taking this approach allows the endpoint to work regardless of whether the `Bearer` prefix has been included or not.

To test the `/resource` endpoint, make a GET request to `http://localhost:8000/resource` and include your JWT in the authorisation header. In your REST client (e.g., Postman), set up the header as follows:

- **Key:** `Authorization`
- **Value:** `Bearer <your-token>`

Replace `<your-token>` with the actual token received from the `/login` endpoint. The format of the header should look something like this:



✓ Authorization Bearer eyJhbGciOiJIUzI1NiIsI...

If the provided token is valid, you should receive a success message. Changing even a single character in the token will cause the verification to fail due to the strict signature check.

Implementing the /admin_resource endpoint

Next, let's create an `/admin_resource` endpoint, restricted to admin users only. Place this code above the `app.listen()` line:

```
// Route to access an admin-only resource, requiring a valid JWT with admin
privileges
app.get('/admin_resource', (req, res) => {
  // Extract the token from the authorisation header
  const authHeader = req.headers['authorization'];
  let token;

  // Check if the authorisation header includes the “Bearer” prefix
  if (authHeader && authHeader.startsWith('Bearer ')) {
    // If it does, split to extract the token part after “Bearer”
    token = authHeader.split(' ')[1];
  } else {
    // If no Bearer prefix, assume the entire header is the token
    token = authHeader;
  }

  // If no token is found, respond with a “401 Unauthorized” error
  if (!token) {
    return res.status(401).send({ error: 'Token missing!' });
  }

  try {
    // Verify the JWT using the secret key
    const decoded = jwt.verify(token, 'jwt-secret');

    // Check if the decoded token has admin privileges
    if (decoded.admin) {
      // If admin privileges are present, respond with a success message
      res.send({ message: 'Welcome, admin! Access granted.' });
    } else {
      // If the user is not an admin, respond with a 403 Forbidden error
      res.status(403).send({
        message: 'Access denied: your JWT is verified, but you lack admin
privileges.'
      });
    }
  } catch (err) {
    // If verification fails, respond with a “401 Unauthorized” error
    res.status(401).send({ error: 'Invalid JWT!' });
  }
});
```

The `/admin_resource` endpoint verifies the JWT in a similar manner to the `/resource` endpoint. If the token is valid, it checks the **`admin`** property in the payload to ensure that the user has admin privileges. If **`decoded.admin`** is **`true`**, access is granted. Otherwise, a “403 Forbidden” status is returned with an explanatory message. If the JWT itself is invalid, a “401 Unauthorized” response is returned.

With the current setup of the **`/login`** endpoint, the token that is generated will not include admin privileges, so accessing **`/admin_resource`** with this token will result in an HTTP 403 error.

Directly attempting to edit the token’s **`admin`** attribute by encoding and re-encoding the JWT will not work, as this will break the signature and make it invalid. The only way to obtain an admin-level token is to modify the **`/login`** endpoint to include **`admin: true`** in the payload and then generate a new token. Update the authorisation header with the admin-level token to access **`http://localhost:8000/admin_resource`** successfully.

Practical task 1

This task aims to help you learn how to encode and decode JWTs.

Follow the steps below to complete this task:

1. Navigate to **jwt.io**.
2. Generate a JWT using jwt.io. Enter HS256 as your algorithm and enter your secret key ("mysecretkey").
3. Replace the payload with the following:

```
{
 "username": "zama",
 "password": "abcdef"
}
```

4. Take a screenshot of the decoded input (payload and secret key) and the encoded output (generated token).
5. Upload the screenshot to this task's folder.

Be sure to place files for submission inside your task folder and click "Request review" on your dashboard.

Practical task 2

Create an Express app by following the steps in the "Implementation of JWTs" section. It should have the following endpoints:

- `/login` – checks a POSTed username and password, and produces a JWT.
- `/resource` – checks the JWT in the auth header and displays a message with the username.
- `/admin_resource` – checks the JWT and displays a message if the token is verified and the token holder is an admin.

Once again, when you are ready to have your code reviewed, delete the **node_modules** folder.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we’ve done a good job?

Click here to share your thoughts anonymously.