Hyperiondev

TASK

Authentication with JWT

Visit our website

Introduction

MAKING SURE YOUR USER IS WHO THEY SAY THEY ARE

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

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact an expert code reviewer to get support on any aspect of your course.

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our expert code reviewers are happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

THE HISTORY OF REST AND EARLY AUTHENTICATION

In the early days of the internet — right when the idea of RESTful requests was first being designed — the software engineers of the time decided that such requests should be stateless. That is, any REST request need not rely on any prior request having been made. This meant that everything a user wanted to do had to happen in one fat request — e.g. to buy a product online the REST request needed to contain enough information to authenticate the user, select the products to buy, and provide payment information.

Even though this meant really long-running requests, there were good reasons for this decision. One is that with the early internet infrastructure, making a request already took a long time, and many of them were bound to fail anyway because of physical networking issues. Another is that the early hardware had a hard time dealing with many requests at a time, and so cutting down on the number of user-requests per transaction was the top priority.

The first web authentication method was created under these limitations and is called basic authentication. It very simply has the username and password in the header of every request. The disadvantage here is that if you're using http (not https) then that password is being sent in plaintext and is rather easily interceptable by methods such as **man-in-the-middle attacks**. It is, however, the simplest and thus the quickest method to implement as a programmer. Basic authentication is still used today for the odd occasion where only a single REST request needs to be made, and the endpoint supports it. For example, getting a weather forecast, stock market history, or version control repo data. Note, however, that it is only ever used over https, and usually not in browser situations (only via APIs in apps, where the user doesn't have access to the requests themselves).

CONTEMPORARY AUTHENTICATION

Today, technology has improved exponentially to the point where you can make 10 requests and with reasonable confidence expect them all to complete before a second has passed — even on mediocre hardware and internet connection by today's standard. 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 endpoint.
2. The auth endpoint checks the data and, if legit, generates an authentication 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.

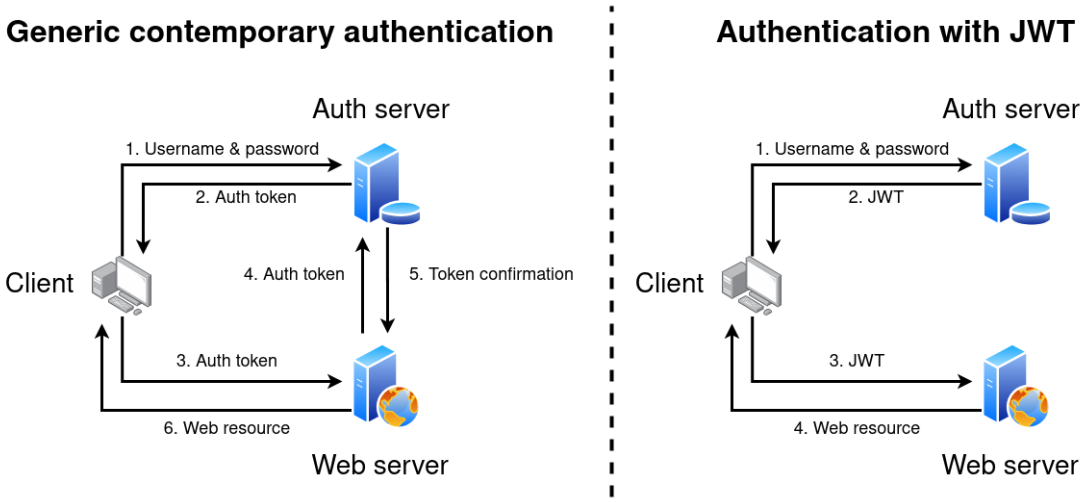
There are many different ways to implement this process. Some have multiple-step authentication, and some require extra internal authentication steps with regular requests. All of them expend a great deal of energy to ensure this generated token is secure, incorruptible, and hacker-proof.

ENTER JWT

In this task, we're going to implement our own way of authentication, using a standard called JSON Web Tokens (JWT). In a nutshell, JWT is a way of representing data that does not allow tampering during transit, and that can be validated using encryption keys. That is, when the server receives a JWT, it has a sure-fire way of determining whether the data it contains is legitimate.

Before we dig into how it works, let's consider the advantages of such a technique. The purpose of authenticating a user is to ensure they only have access to the resources they should. For example, in a WordPress site, an admin should be able to delete and re-categorise posts, while authors should only be able to create and edit posts. If you're using a generic auth method from above, the server would have to check the auth token against the database to ensure it is valid, and then do another lookup to check what resources the user has access to. This is a very time-consuming procedure, especially if your auth endpoint is on another server entirely (which they usually are on big projects).

JWT circumvents many of these issues by allowing the auth endpoint to put all the relevant permission data *inside* the token. When the server receives it, it can validate and decode the token (without any DB lookups) and immediately grant or deny access. Compare the two kinds of authentication methods below:



You'll notice that with JWT there is no direct communication between the web server and the auth server. This is fine because the JWT contains all the necessary auth data and its legitimacy can be validated on its own. Let's look into how exactly this is possible.

THE INNER WORKINGS OF JWT

A JSON Web Token is at the end of the day just a string: a string that consists of three parts, namely the *header*, *payload*, and *signature*. Each part is separated by a dot and is base64-encoded. Base64-encoding is a way of representing any kind of data in text format. In the context of JWT, it's simply used to represent a lot of data in a small text space. So, the just of a JWT will look something like this (though typically a bit longer):

```
abc123.def456.ghi789
```

The header part is just a JSON object describing the token. Specifically, it mentions the token type (JWT) and the algorithm to be used for signing. We'll be specifying `HS256`, which is short for **HMAC-SHA256**, a very common hash-encryption standard and the one we'll be using in this task.

The header before base64-encoding:

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

And after:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

The payload is a JSON object with any content of your choosing. Because we're using JWT for authentication, we'll add some basic user and permission information. There exists a list of key-name standards for JWT payloads **here**, but it's not crucial that you follow them since it's just your own backend code that will be interpreting the data.

Example payload before base64-encoding:

```
{
  "id": 1234,
  "name": "John Doe",
  "admin": true
}
```

And after:

```
eyJpZCI6MTIzNCwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
```

Finally, we need to produce the signature. This is done using the algorithm specified in the header. In our case, HMAC-SHA256 — a complicated name that really just represents a kind of hashing function. You may recall hashing from the data structures task. It's exactly the same idea here: an object (message or index) is one-way encoded to a seemingly random hash. Though it's not completely random, because the hash will reliably always be computed the same given the same input. Our HS256 algorithm takes this a step further by also requiring a secret key during encoding. So in this case, to produce the signature you need the message (the header and the payload) and also a secret key. The combination of the three produces a hash (the signature) that will always reliably compute the same given the same header, payload, and key.

It's important that the signature algorithm always produces the same signature when given the same input because that's how we validate the JWT. Here's how to get a signature in pseudocode:

```
header = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9'
```

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

You needn't worry about how the `HS256` method works - these kinds of encoding and encryption functions are written by experts in their fields, and it's best we just use their libraries instead of reinventing the wheel. Here's what the signature would be in our example:

```
q_SJ2dKnaN8sjVtwi5G6yCJ6CpU-OSiiFR-ZBNm0G64
```

And so the final JWT would be this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNCwibmFtZSI6IkpvaG4gRG9
lIiwiYWRtaW4iOnRydWV9.q_SJ2dKnaN8sjVtwi5G6yCJ6CpU-OSiiFR-ZBNm0G64
```

Now this token may look like garbage, but it actually contains valuable data (the user's id, name, and admin-level). This can be read by base64-decoding the middle part of the token (the payload). Note that base64 encoding is two-way and does not require a secret key. That means that anyone with the token can read its contents. This is why you should never put sensitive data like passwords or credit card numbers in JWTs. Permission and user data is fine, though.

You may wonder what the point of all this encryption and encoding is if anyone can see the data. The purpose of JWTs is not to hide data, but rather to ensure its integrity. That is, to ensure it cannot be tampered with. This is achieved with the signature, for which you need a secret key to produce. So if anyone alters the payload of the key, they will be unable to produce the correct signature unless they have the secret key.

This is the value of JWT. When the server receives a token it can verify its integrity and legitimacy and therefore trust its contents. No matter what the payload, if the signature is correct, the server can be certain that it was produced by the authentication endpoint.

For interest's sake, there's a website called **jwt.io** that contains more information about JWT best practices as well as an encoder-decoder so that you can check that you're producing correct tokens.

JWT IRL

Enough theory. Let's put JWT into practice by creating an Express app. Use your knowledge from the earlier Express tasks to create a new Express app with the main file: **index.js**. In it, place the following code to get a basic app running.

```
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
const port = 8000

app.use(bodyParser.json())

app.post('/login', (req, res) => {
   const usr = req.body.username
   const pwd = req.body.password
   res.send(`Username: ${usr}\n Password: ${pwd}`)
})

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

If you haven't yet, be sure to install Express and its accompanying body parser with the following command.

```
npm install express body-parser
```

Start the app with:

```
node index.js  # or user nodemon if you prefer
```

Now using Postman, or any other REST testing platform, make a POST request to `http://localhost:8000/login` with the following body:

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

Be sure to include the header `Content-Type: application/json`, so that the body's content type is accurately described. This is standard practice in 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 hello-world Express app working, let's discuss our design. We're going to have two endpoints: an authentication endpoint and a resource endpoint. The auth endpoint will do password validation and will respond with a JWT if successful. The resource endpoint will respond with an arbitrary resource if the JWT is legit. We'll get into specifics later.

Let's craft the authentication endpoint. The first thing to do is to check the username and password, and respond with HTTP 403 if it's wrong — this is the standard response code for auth failure in RESTful APIs. Remove the `res.send()` statement from before and replace it with the following.

```
if (usr==='zama' && pwd==='secret'){
   //todo
}else{
   res.status(403).send({'err':'Incorrect login!'})
}
```

In a production application, you would obviously not hardcode the username and password, but rather look it up from a database. We're just doing it for demonstrative purposes.
Also, note that the error response is in JSON format. This isn't strictly necessary (it could have just been a simple string), but it's good practice to keep request and response content type consistent throughout your entire app.

Restart your app if you're not using nodemon. Making the same request, you should now see the incorrect login error message.

The most popular JWT library for node is simply called `jsonwebtoken`. Install it the usual way and add the following line to the top of your **index.js** file.

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

Now replace the todo-comment from above with the following code that generates a proper JWT.

```
payload = {
   'name': usr,
   'admin': false
}
const token = jwt.sign(JSON.stringify(payload), 'jwt-secret',
   {algorithm: 'HS256'})
res.send({'token': token})
```

See how easy the library makes the JWT generation process? We didn't have to code up any of the sticky algorithms we spoke about above.

Restart your app if necessary. Change your request body to have the correct password and make the request again. You should now receive your token:

```
{
"token":"eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiemFtYSIsImFkbWluIjpmYWxzZX0.KT
o5xXD2ecZWEaABPny6Y2Z6dM0AxPcdtWAAW_TcKTM"
}
```

Now we'll code up the resource endpoint. Add the following to your **index.js**.

```
app.get('/resource', (req, res) => {
   const auth = req.headers['authorization']
   const token = auth.split(' ')[1]
   try {
       const decoded = jwt.verify(token, 'jwt-secret')
       res.send({'msg':
     `Hello, ${decoded.name}! Your JSON Web Token has been verified.`})
   }catch (err) {
       res.status(401).send({'err': 'Bad JWT!'})
   }
})
```

- JWTs are usually added to the Authorization header of requests, in the format:
  `Authorization: Bearer <token>`
  This is the REST standard. The first two lines take care of extracting said token. Note that because of the way Express processes headers, the auth header is accessed with a lowercase "a", even though the header itself has an uppercase "A".
- The `jwt.verify()` method verifies a given token using the specified secret key. If it produces an unexpected signature (due to a bad key or tampered

token) or if the token is malformed, an error is thrown. We don't have to specify the algorithm here because it's already in the header of the token.
- If verification is successful, the token can be trusted and its payload is decoded. We then use this to construct a personalised message for the user.
- If the verification fails, an HTTP 401 status is returned. This is the REST standard for bad authorisation.

Now in your REST tester, make a GET request to `http://localhost:8000/resource`, and add the following header to it (as one line):

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiJ9.eyJuYW1lIjoiemFtYSIsImFkbWluIjpmYWxzZX0.KTo5xXD2ecZ
WEaABPny6Y2Z6dM0AxPcdtWAAW_TcKTM
```

Your request should succeed. Notice how if you change merely a single character in the token, the request fails. This is due to the strictness of the signature algorithm.

Finally, let us make use of user permissions. Add an admin resource and verify the user like so:

```
app.get('/admin_resource', (req, res) => {
   const token = req.headers['authorization'].split(' ')[1]
   try {
       const decoded = jwt.verify(token, 'jwt-secret')
       if (decoded.admin){
           res.send({'msg': 'Success!'})
       }else{
           res.status(403).send(
           {'msg': 'Your JWT was verified, but you are not an admin.'})
       }
   }catch (e) {
       res.sendStatus(401)
   }
})
```

Making a request to this endpoint with your previous token will result in an HTTP 403 because even though the token was verified, your user is not an admin. You'll note that attempts to change the payload's admin-attribute to true directly (using base64 decode and encode) will result in a token verification failure. This is because the signature will be incorrect. The only way to get an admin token is to have your auth endpoint dispense it as such. To see this in action, change your login

endpoint to have its payload contain `admin: true`, and use the resulting token to request to `http://localhost:8000/admin_resource`.

Compulsory Task 1

Create an Express app by following the guide. It should have the following endpoints:

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

Compulsory Task 2

Extend your Express app to have more comprehensive authorisation.

- Create 3 users (store them as you please — hardcoded, in a file, or in a db) with the following route access permissions:
  - Mazvita - `/a`
  - Meagan - `/a` and `/b`
  - Kabelo - `/b` and `/c`
- The users should only have permission to access the routes mentioned above, and the `/login` endpoint. You may store these permissions in your JWT as you see fit. Remember, anything you can put in JSON, you can put in the JWT payload.
- Create the `/a`, `/b`, and `/c` endpoints, each checking if the requesting user has permission to access it.

Submit your **index.js**.

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.