

the Master Course

{C0DENATION}

Backend Development

JWT Tokens



Learning Objectives

To know what a Route is.

To understand the basics of Client/Server requests and responses

To know what a database is and what it does.

To be able to make an Express static server.

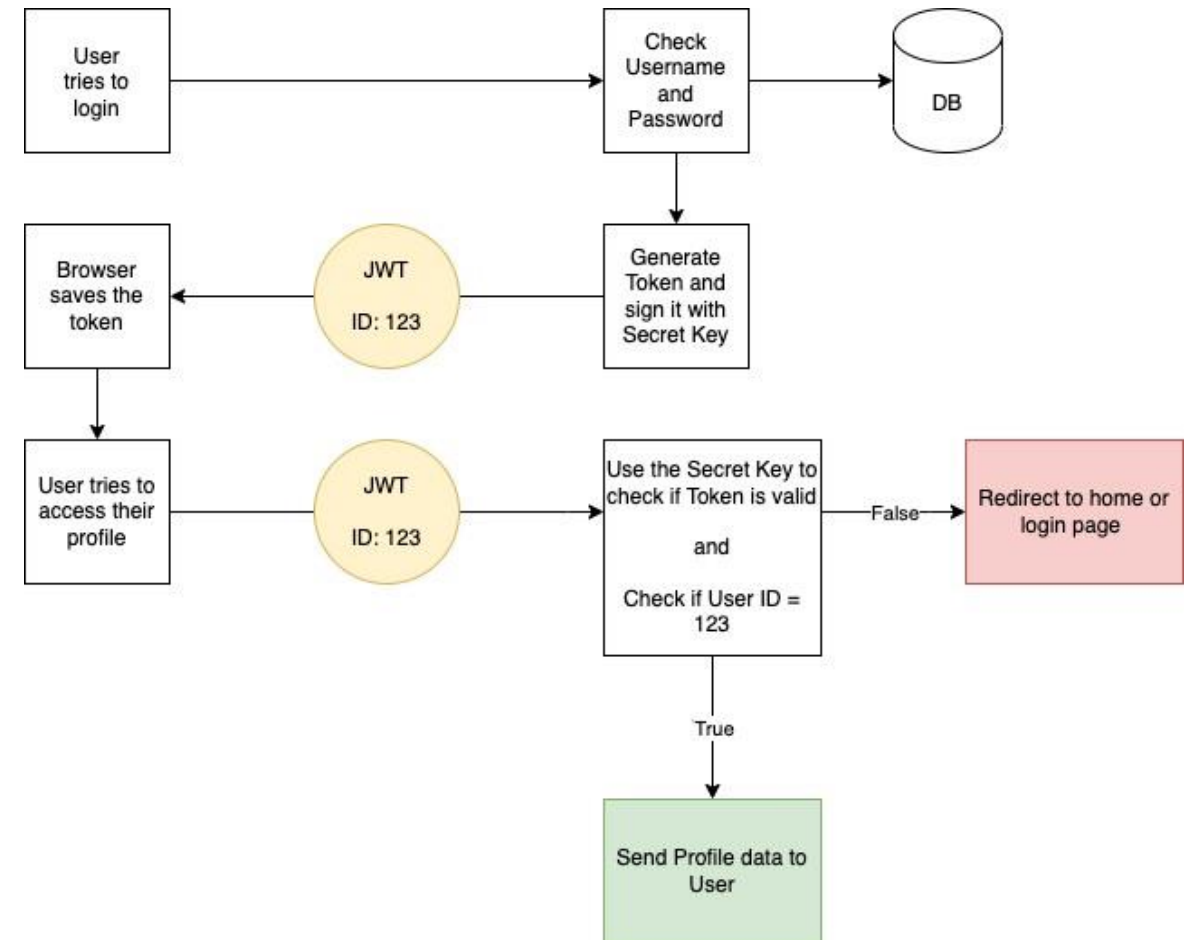
What is JWT?

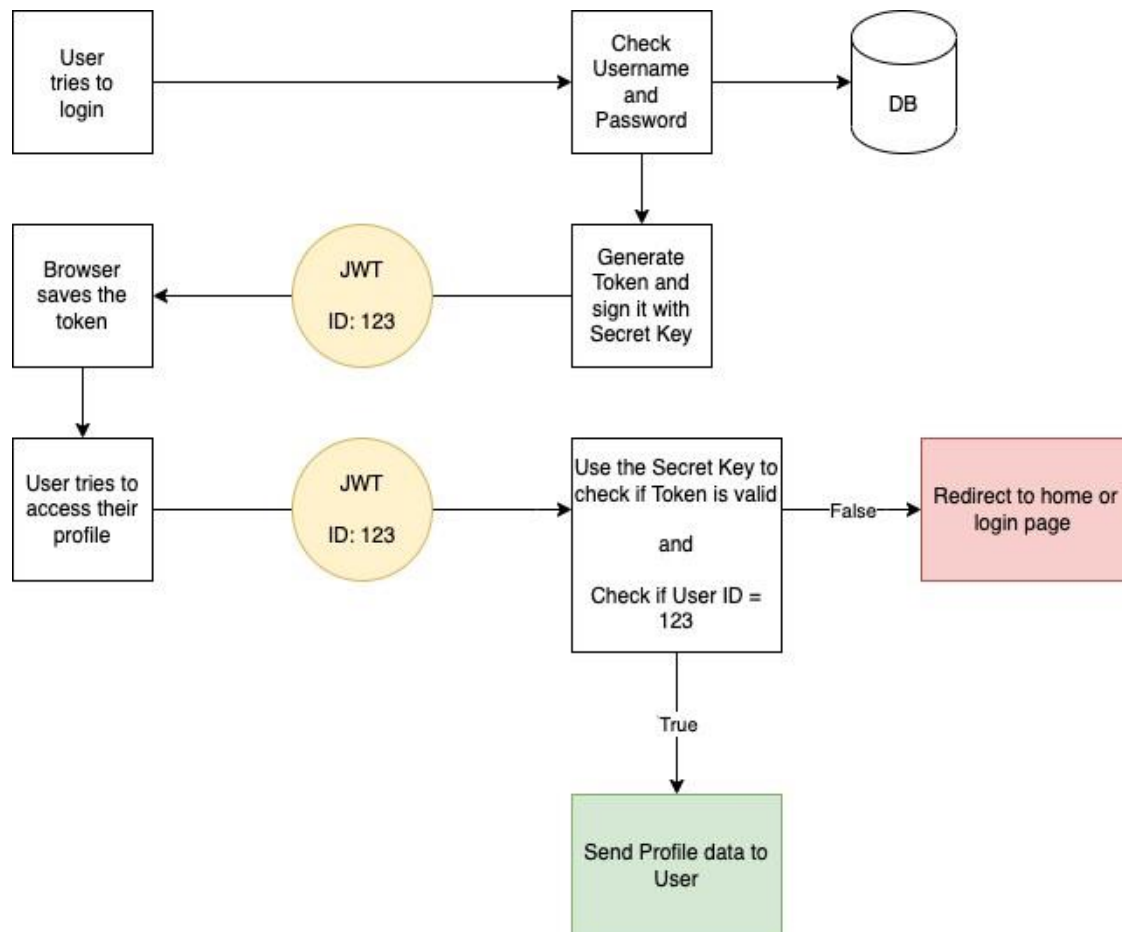
JWT (JSON Web Token) is an open standard used to share security information between client and server.

JWTs are mostly used to authenticate users in websites.

Let's say we have a Web Application called Tweeter. Tweeter can do a bunch of fancy stuff but let's focus on the user profile.

Each user should only be able to access their own user profile settings, e.g., a user with ID: 123 should only be able to access profile settings of User ID: 123.





Two things to notice:

1. Once the User/Browser has the Token, the DB is no longer needed to authenticate the user. (stateless authentication).

2. With the Token, we can send information about the user (e.g., the ID).

This information can then be verified using the signature.

More on that later...

JWT tokens are sent in the HTTP Authorization request header

These headers are used to provide credentials that authenticate a user when a request is made - allowing access to a protected resource.

JWT's are encoded when they are sent. This makes them easy to decode and check the information stored inside of them.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2Mjc5MjgyMzksInVzZXJSb2x1cyI6WyJBRE1JTlIsIkRCX1JFQUQiLCJlEQ19XUk1URStJdLCJ1c2VySWQiOiJlYmZQ1LCJpYXQiOiJlMjc5MjgyMT19.8vTwsB0p8LSa0sdc0nWAUnmWAAg0nS0E1B3bfaiSRfQ

This is nothing more than encoded JSON. Notice that it is encoded, not encrypted. This means that we can easily decode it.

Using the jwt.io website we can easily
decode a **JWT** to understand its structure

The JWT is made up of three parts...

:

The image shows a web-based JWT decoder interface. On the left, under the heading "encoded", there is a text area containing a Base64-encoded JWT token. On the right, under the heading "Decoded", the token is broken down into three parts: Header, Payload, and Footer. The Header shows the algorithm as HS256 and the token type as JWT. The Payload contains user information including an expiration time, roles, and a user ID. The Footer shows the HMACSHA256 signature formula and a checkbox to toggle between secret and base64 encoding.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE2Mjc5MjgyMzksInVzZXJSb2xlcyl6WyJBRE1JTlIsIkRCX1JFQUQiLCJlYX0iOiE2Mjc5MjgyMT19.8vTwsB0p8LSa8sdc0nWAUnmWAAG0nSBE1B3bfaiSRFQ
```

Decoded

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{  "exp": 1627928239,  "userRoles": [    "ADMIN",    "DB_READ",    "DB_WRITE"  ],  "userId": 12345,  "iat": 1627928119}
```

VERIFY SIGNATURE

```
HMACSHA256(  base64UrlEncode(header) + "." +  base64UrlEncode(payload),  your-256-bit-secret)
```

☐ secret base64 encoded

Header

Information on the signing algorithm that was used.

Payload

The user information, e.g., userId and any other information you want to use.

Footer

The signature of the token that can be verified with the Secret Key. If the Token gets changed in any way, it won't be valid anymore.

What is a JWT Signature?

The signature is used to verify the sender of the JWT and to ensure that the message wasn't changed along the way.

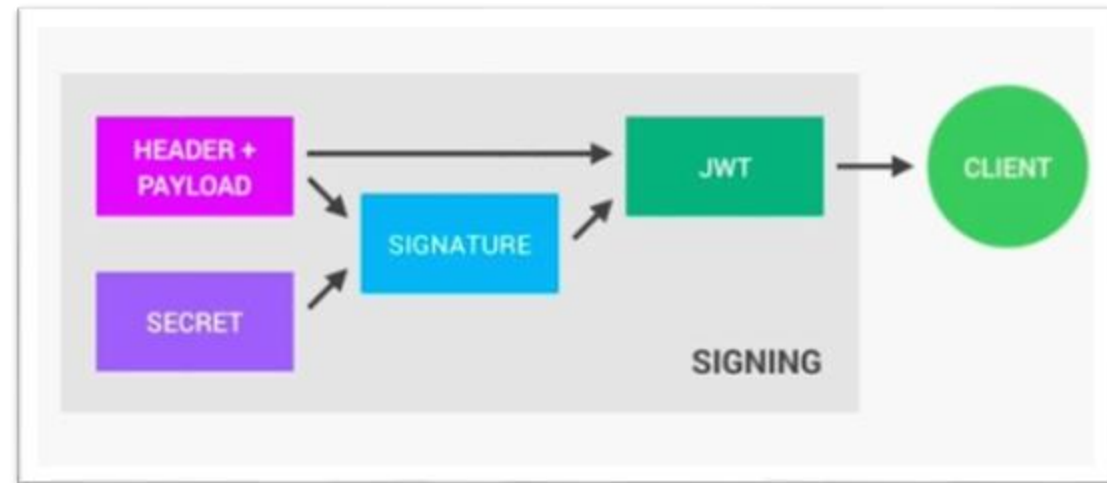
This whole process is called signing the Json Web Token. The signing algorithm takes the header, payload, and the secret key to create a unique signature for each JWT.



Then together with the header and the payload, the signature forms the JWT, which then gets sent to the client.

Once the server receives a JWT to grant access to a protected route, it needs to verify it in order to determine if the user really is who they claims to be.

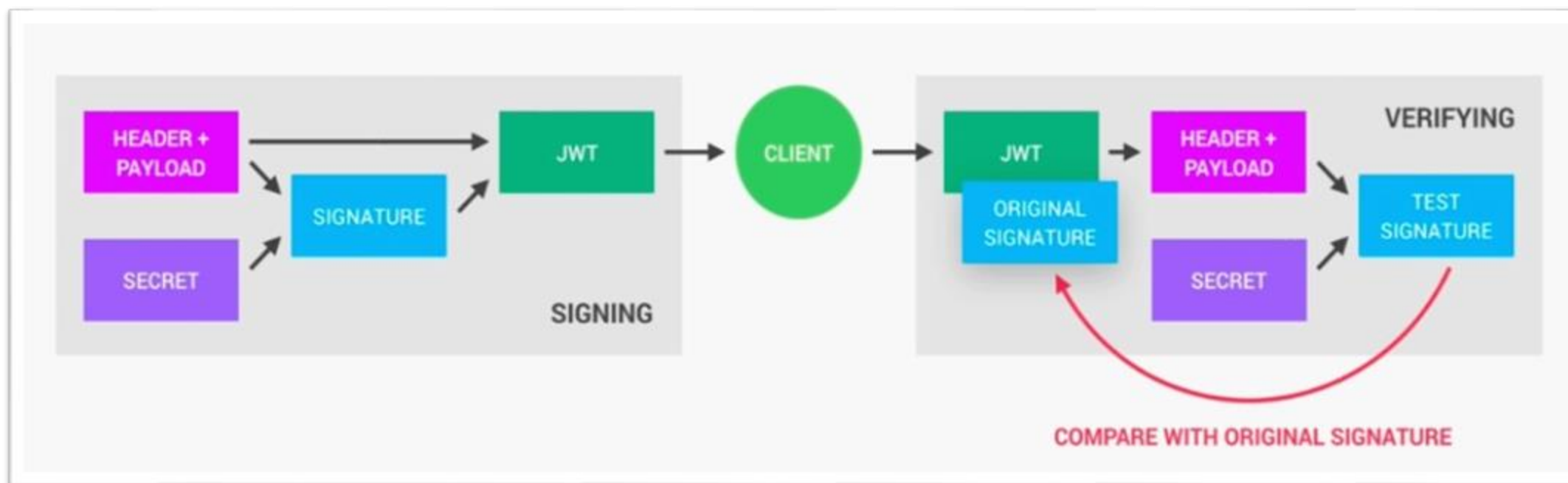
In other words, it will verify if no one changed the header and the payload data of the token.



So, how does this verification actually work?

Once the JWT is received, the verification will take its header and payload, and together with the secret that is still saved on the server, basically create a test signature

But the original signature that was generated when the JWT was first created is still in the token, right? And that's the key to this verification.



If we compare the test signature to the original signature and they are the same, then it means that the payload and the header have not been modified.

Now let's do some practice with node.js

In order to generate JWT Tokens in our REST API, we will use an NPM Library called jsonwebtoken.

First things first: let's install jsonwebtoken

```
admin2@CN18s-MacBook-Pro m43-rest-api % npm i jsonwebtoken
```

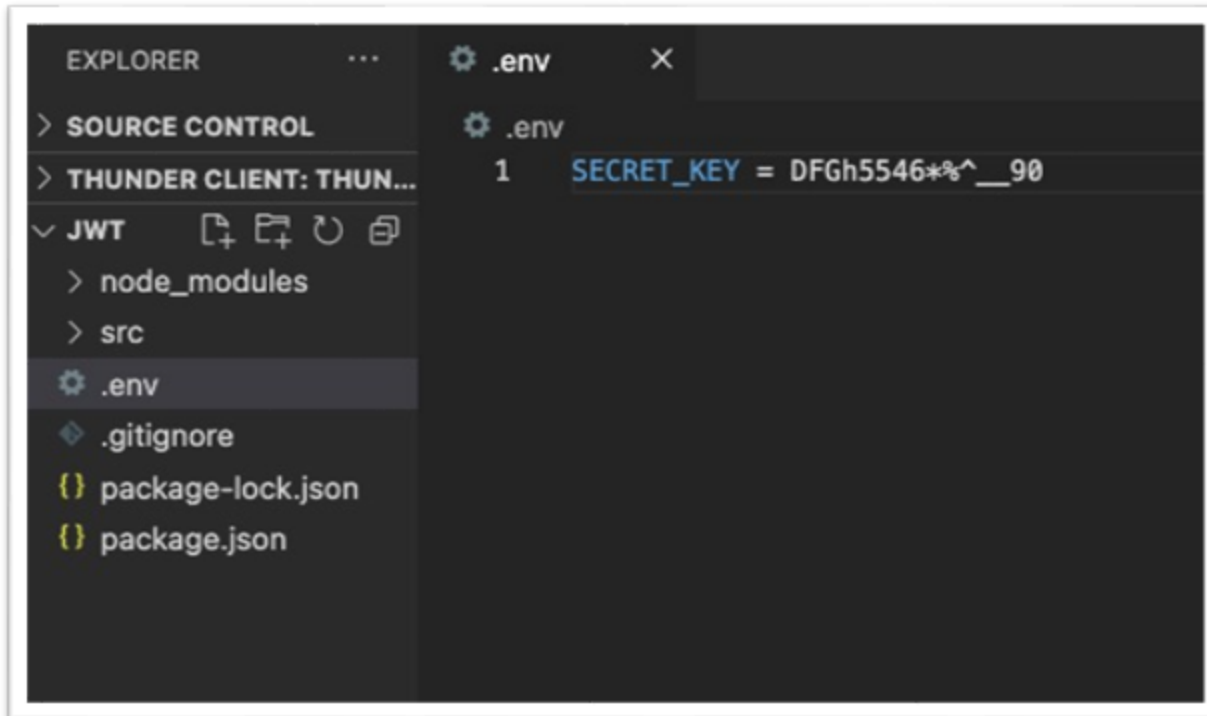
Once installed, jsonwebtoken should now appear in the dependencies in your package.json file on

Now we need to import jsonwebtoken into the specific files where you need it

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

```
"dependencies": {  
  "bcrypt": "^5.1.0",  
  "cors": "^2.8.5",  
  "dotenv": "^16.0.3",  
  "express": "^4.18.2",  
  "jsonwebtoken": "^9.0.0",  
  "mysql2": "^3.2.0",  
  "nodemon": "^2.0.21",  
  "sequelize": "^6.29.0"  
}
```

JWT



The image shows a screenshot of the Visual Studio Code interface. On the left, the Explorer sidebar is open, showing a project structure with folders 'node_modules' and 'src', and files '.env', '.gitignore', 'package-lock.json', and 'package.json'. The '.env' file is selected. The main editor area displays the contents of the '.env' file, which includes the line 'SECRET_KEY = DFGh5546*%^__90'.

```
EXPLORER
> SOURCE CONTROL
> THUNDER CLIENT: THUN...
JWT
  > node_modules
  > src
  .env
  .gitignore
  {} package-lock.json
  {} package.json

.env
1 SECRET_KEY = DFGh5546*%^__90
```

Next we need to create a secret key to sign our JWTs. We store the secret as an environment variable and it is loaded in when required, rather than hard coding the secret key into our code. This way our key is more secure when a token needs to be verified and decoded.

Jsonwebtoken .sign() Method

This method is used to generate and sign JWT Tokens.

JWT

Here we have a function that generates and signs JWTs. The `.sign()` method takes two values. The first is an object with the values we want to store in our token.

```
require("dotenv").config();
...
const jwt = require("jsonwebtoken")

const generateAndSignJWT = () => {
  const userId = 123
  const admin = true

  const token = jwt.sign({ id: userId, isAdmin: admin }, process.env.SECRET_KEY);
  console.log(token)
}

generateAndSignJWT()
```

In this case the values stored in the `userId` and `admin` variables are used. The second value is the secret key, which is used to sign the JWT. We load this from our environment variables for security.

The encoded token is then logged in the terminal.

```
admin2@M18s-MacBook-Pro jwt % node src/index.js
Generated Token -
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzLCJpdiI6dWVudCJpYXQiOiJlZ200A3MDg0MTh9.jYzwLwosThdHv4QYD8_GpDQYTSS10jT35A2vk5zeY4
```

Once generated, the token is added to the response along with the username and email of the user who is logging in.

```
const login = async (req, res) => {
  try {
    const token = await jwt.sign({ id: req.user.id }, process.env.SECRET);

    res.status(201).json({
      message: "success",
      user: {
        username: req.user.username,
        email: req.user.email,
        token: token,
      },
    });
  } catch (error) {
    res.status(501).json({ errorMessage: error.message, error: error });
  }
};
```

Jsonwebtoken.verify () Method

This method is used to verify the signature of a token.

```
let generatedToken = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX" +  
"VCJ9.eyJpZCI6MTIzLCJpc0FkbWluI" +  
"jp0cnVlLCJpYXQiOiE2ODQ3ODQwOTV9." +  
"vWV4K1XBBrubMQXVjvPPbvVs_3R8CVyg7w_2k1cCGLI"  
  
let otherToken = 'Random String'  
  
const verifyToken = () => {  
  try {  
    const decodedToken = jwt.verify(generatedToken, process.env.SECRET_KEY);  
    console.log(decodedToken)  
    console.log("Vaild Token")  
  } catch {  
    console.log("invaild token")  
  }  
}  
  
verifyToken()
```

The `.verify()` method takes two values. The first is a previously generated token. The second value is the secret key, which is used to sign the JWT. We load this from our environment variables for security.

If they match, then the `decodedToken` is logged to the terminal. Otherwise "Invalid token is logged".

```
admin2@CN18s-MacBook-Pro jwt % node src/index.js  
{ id: 123, isAdmin: true, iat: 1680784095 }  
Vaild Token
```

`decodedToken` is an object with the values encoded into the JWT when it was signed and the time the JWT was generated.

Here is the same function again but this time the otherToken value is passed to the .verify() method.

```
let generatedToken = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpX" +  
"VCJ9.eyJpZCI6MTIzLCJpc0FkbWluI" +  
"jp0cnVlLCJpYXQiOi0jE20DA30DQw0TV9." +  
"vwV4K1XBBruBMQXVjvPPbvVs_3R8CVyg7w_2k1cCGLI"  
  
let otherToken = 'Random String'  
  
const verifyToken = () => {  
  try {  
    const decodedToken = jwt.verify(otherToken, process.env.SECRET_KEY);  
    console.log(decodedToken)  
    console.log("Vaild Token")  
  } catch {  
    console.log("invaild token")  
  }  
}  
verifyToken()
```

The otherToken doesn't contain the secret key or was encoded so invalid token is logged.

```
admin2@CN18s-MacBook-Pro jwt % node src/index.js  
invaild token
```



```
const tokenCheck = async (req, res, next) => {
  try {
    const token = req.header("Authorization");

    const decodedToken = await jwt.verify(token, process.env.SECRET);

    const user = await User.findOne({ where: { id: decodedToken.id } });

    if (!user) {
      const error = new Error("User is not authorised");
      res.status(401).json({ errorMessage: error.message, error: error });
    }

    req.authCheck = user;

    next();
  } catch (error) {
    res.status(501).json({ errorMessage: error.message, error: error });
  }
};
```

Firstly the encoded token is retrieved from the authorization header, Again `.verify()` takes two values. The first is our encoded token. The second value is the secret key, which was used to sign the JWT. We load this from our environment variables for security.

Next we use the `userId` that was encoded into the original JWT to try and find that user in our database using the `.findOne()` method. If the user is not found or the token is invalid `.findOne()` will return false, meaning the user doesn't exist and an error is thrown and sent in the response.

If the token is verified and the user exists, `next()` is called and the request can continue.

Learning Objectives

To know what a Route is.

To understand the basics of Client/Server requests and responses

To know what a database is and what it does.

To be able to make an Express static server.