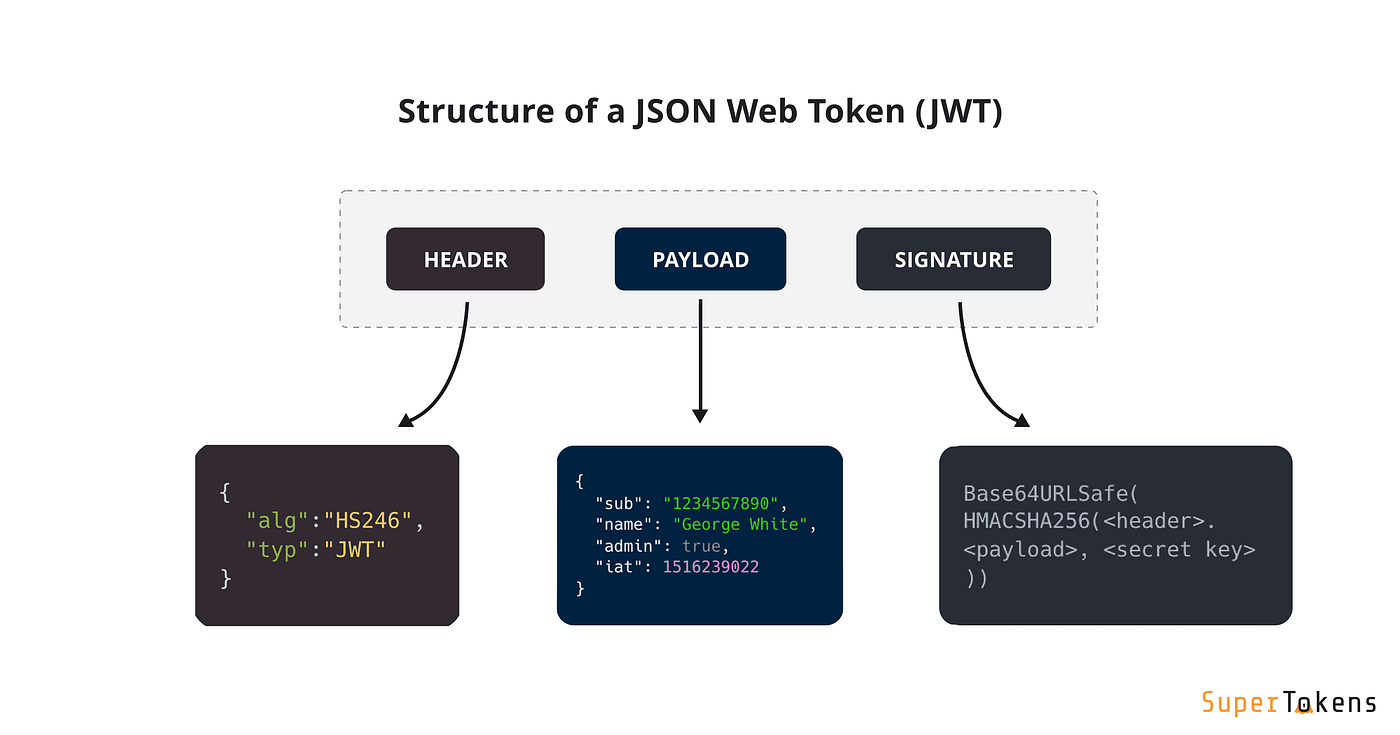
**DAY 10**

As we know tokens are generic authentication credentials or authorization tokens used to authenticate users or authorize access to resources in web applications.

Tokens refer to a broad category of authentication credentials, whereas JWT is a specific format or implementation of tokens.

### *JWT ( JSON Web Token )*

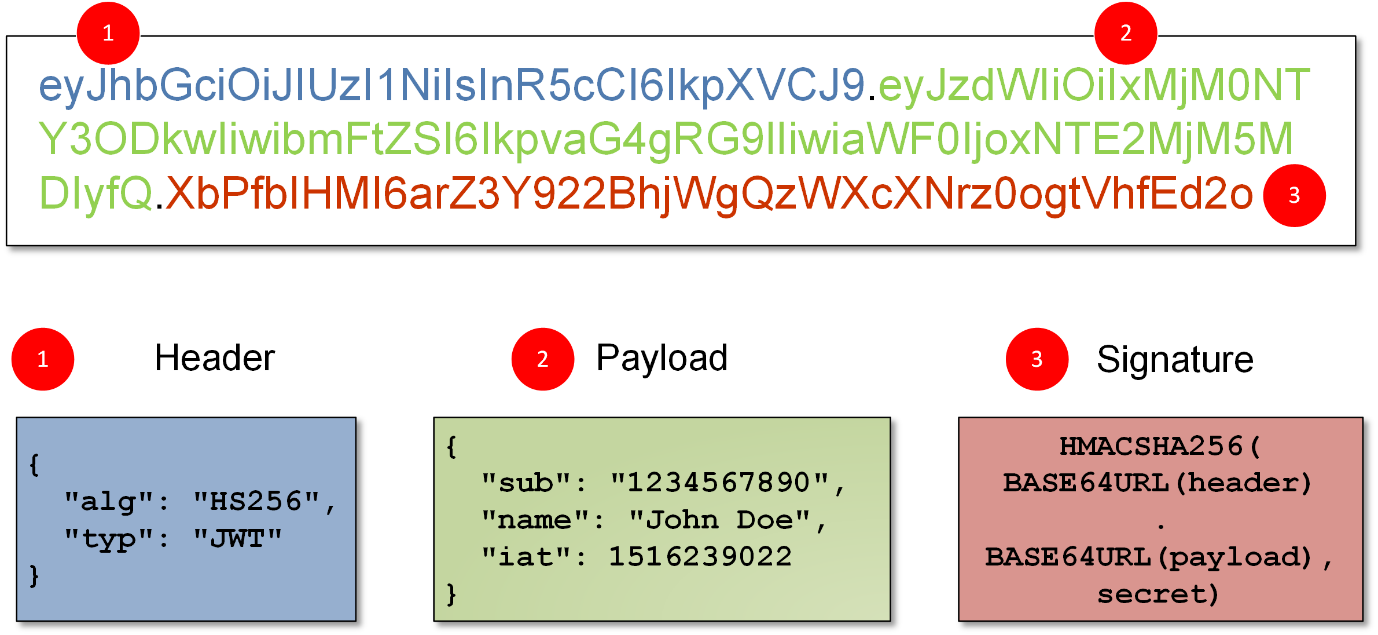
* **Definition:** JWT is a specific type of token format defined by the JSON Web Token standard (RFC 7519). It is a compact and self-contained means of transmitting information between parties as a JSON object.
* **Structure:** JWTs consist of three parts: header, payload, and signature. They are typically encoded and signed using cryptographic algorithms.
* **Usage:** JWTs are commonly used for authentication and authorization in web applications and APIs. They can store user claims, such as user ID, roles, permissions, and custom data, in a secure and portable format.
* **Statelessness:** JWTs are stateless, meaning the server does not need to store session information. This makes them suitable for distributed architectures and scalable systems.



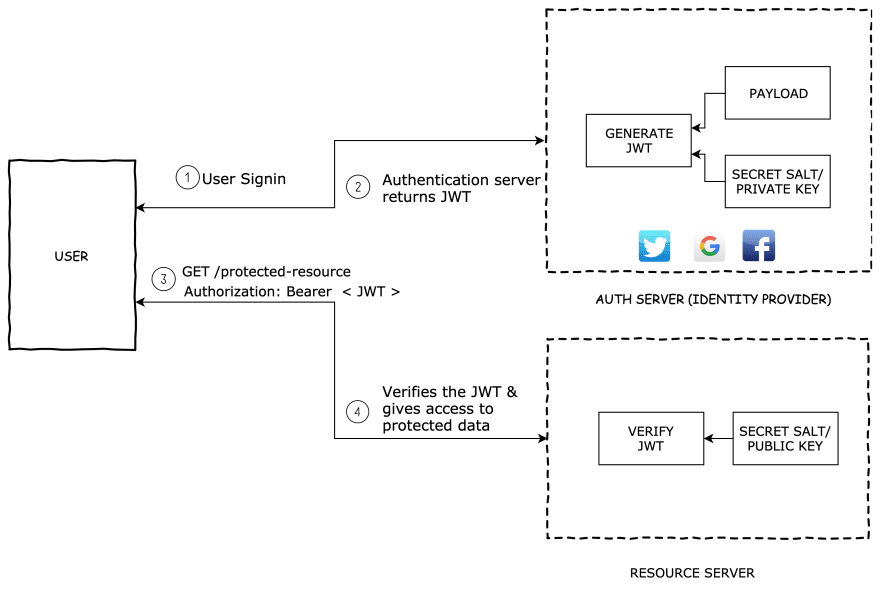
### *JWT Structure*

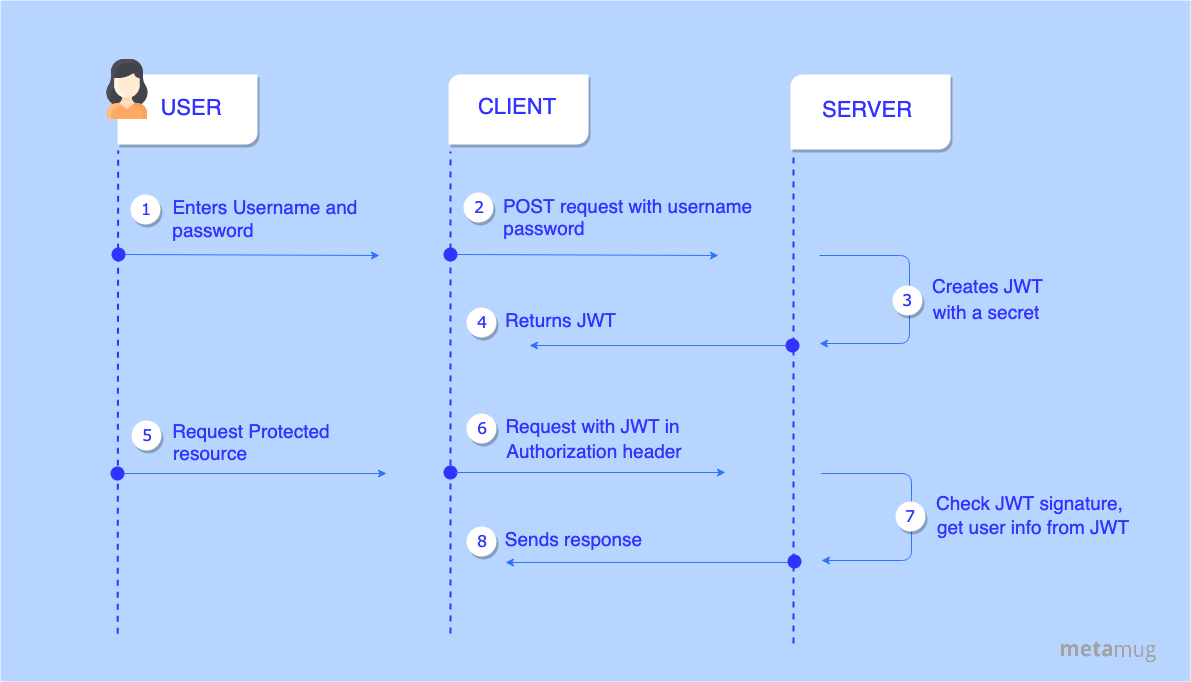
A JWT is composed of three sections separated by dots (.), following the format header.payload.signature.

1. **Header**: Contains metadata about the type of token and the cryptographic algorithms used to secure it. It typically consists of two parts:
   * **Typ (Type)**: Specifies the type of token, usually set to "JWT".
   * **Alg (Algorithm)**: Indicates the cryptographic algorithm used to sign the token, such as HMAC SHA256 or RSA.
2. **Payload**: Contains the claims or statements about the subject (user) and any additional data. It consists of a set of claims that represent assertions about the user, such as their identity, roles, or permissions. Claims are categorized into three types:
   * **Reserved Claims**: Predefined claims standardized by the JWT specification, such as iss (issuer), sub (subject), aud (audience), exp (expiration time), and iat (issued at).
   * **Public Claims**: Custom claims defined by the application developer to convey information about the user.
   * **Private Claims**: Custom claims agreed upon by parties that exchange JWTs, not registered or standardized.
3. **Signature**: Verifies the integrity of the token and ensures that it has not been tampered with during transmission. It's created by taking the encoded header, encoded payload, a secret (for HMAC algorithms), and applying the specified algorithm to generate the signature.



### *Authentication Flow*





### *JWT Functions*

Certainly! In the context of JSON Web Tokens (JWT), jwt.sign() and jwt.verify() are two crucial functions provided by the jsonwebtoken library in Node.js. Here's what they do

**jwt.sign()**:

* This function is used to generate a new JWT token based on the provided payload and options.
* It takes three parameters:
  + payload: This is the data you want to include in the token. It can be any JSON object containing user information, metadata, or any other relevant data.
  + secretOrPrivateKey: This is the secret key used to sign the token. It can be a string or a buffer containing a secret cryptographic key.
  + options (optional): These are additional options that control the behavior of the token generation process, such as expiration time (expiresIn), algorithm (algorithm), and more.

| const jwt = require('jsonwebtoken');  // Payload containing user information const payload = { userId: '123456', username: 'exampleuser' };  // Secret key for signing the token const secretKey = 'your\_secret\_key';  // Generate a new JWT token const token = jwt.sign(payload, secretKey, { expiresIn: '1h' }); |
| --- |

**jwt.verify()**:

* This function is used to verify and decode a JWT token to retrieve the original payload.
* It takes three parameters:
  + token: The JWT token to be verified.
  + secretOrPublicKey: The secret key or public key used to verify the token's signature. If the token was signed using a symmetric algorithm (e.g., HMAC), you need to provide the same secret key used to sign the token. If it was signed using an asymmetric algorithm (e.g., RSA), you need to provide the public key corresponding to the private key used for signing.
  + options (optional): Additional options for verification, such as algorithms (algorithms), audience (audience), issuer (issuer), and more.

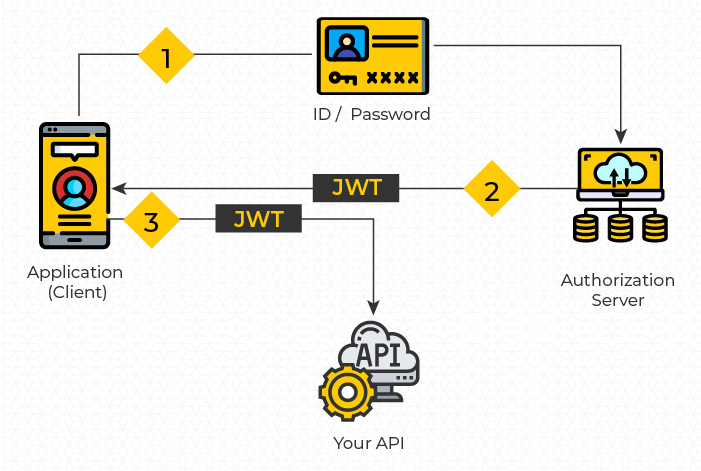
| // Verify and decode the JWT token jwt.verify(token, secretKey, (err, decoded) => {  if (err) {  // Token verification failed  console.error('Token verification failed:', err.message);  } else {  // Token verification successful  console.log('Decoded token:', decoded);  } }); |
| --- |

### 

### *Our Hotel App Flow*

For the First Time, User is completely new to the site

1. **Signup Route (/signup):** This route will handle user registration and issue a JWT token upon successful registration.
2. **Login Route (/login):** This route will handle user login and issue a new JWT token upon successful authentication.
3. **Protected Routes:** These routes will be accessed only by providing a valid JWT token.



### *Install JWT*

First, you'll need to install the necessary packages for working with JWT. In Node.js, you can use packages like jsonwebtoken to generate and verify JWTs.

| npm install jsonwebtoken |
| --- |

### *jwtAuthMiddleware*

Create a JWT Auth Middleware function, which is responsible for authentication via Tokens

| // jwtAuthMiddleware.js const jwt = require('jsonwebtoken');  const jwtAuthMiddleware = (req, res, next) => {  // Extract the JWT token from the request header  const token = req.headers.authorization.split(' ')[1];  if (!token) return res.status(401).json({ error: 'Unauthorized' });   try {  // Verify the JWT token  const decoded = jwt.verify(token, process.env.JWT\_SECRET);  // Attach user information to the request object  req.user = decoded;  next();  } catch (err) {  console.error(err);  res.status(401).json({ error: 'Invalid token' });  } };  module.exports = jwtAuthMiddleware; |
| --- |

* We can certainly change the variable name from req.user to req.EncodedData or any other name you prefer. The choice of variable name (EncodedData, userData, etc.) is flexible and depends on your application's conventions or requirements.
* The key aspect is to make sure that the decoded user information is attached to the request object (req) so that it can be easily accessed by other middleware functions or route handlers.

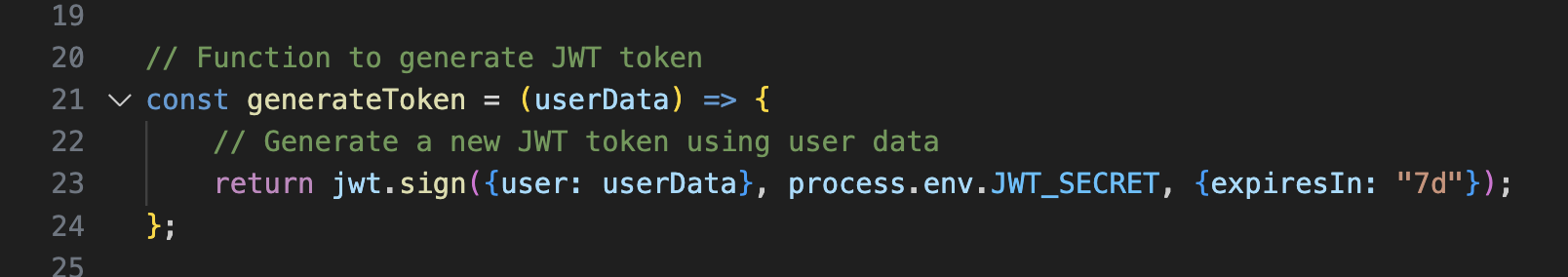
Once you've attached the decoded user information to req.EncodedData within the jwtAuthMiddleware, you can access it in further routes or middleware functions in your application

| // Example route using req.EncodedData router.get('/profile', jwtAuthMiddleware, (req, res) => {  // Access the decoded user information from req.EncodedData  const userData = req.EncodedData;   // Now you can use userData to access user properties like username, role, etc.  res.json({ username: userData.username, role: userData.role }); }); |
| --- |

* Now, Let’s create a JWT token to generate functions

| // Function to generate JWT token const generateToken = (userData) => {  // Generate a new JWT token using user data  return jwt.sign({user: userData}, process.env.JWT\_SECRET, { expiresIn: '1h' }); };  module.exports = { jwtAuthMiddleware, generateToken }; |
| --- |

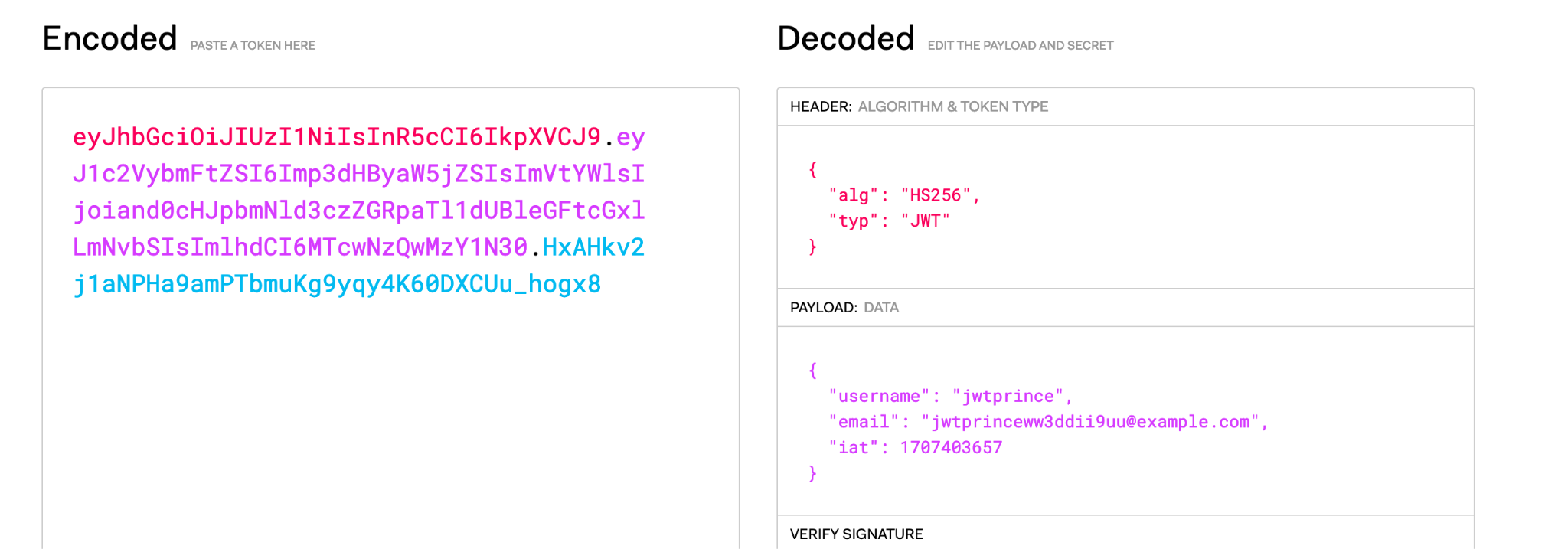
The expiresIn option of jwt.sign() expects the time to be specified in seconds or a string describing a time span, such as '2 days' or '10h'.

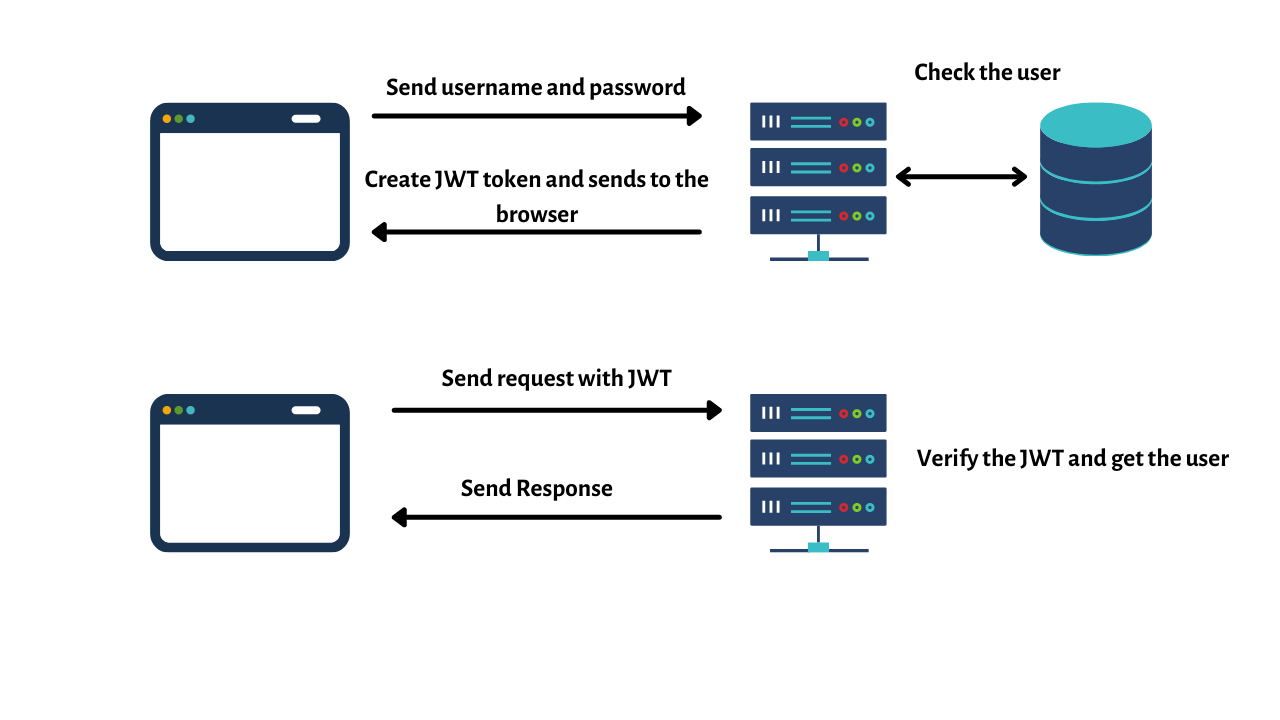


* If you want to make sure, expireIn parameter works properly then you have to pass the payload as a proper object.
* Your payload needs to be an object otherwise it's treated as a string.



### *Validate JWT in Encoder*





### *Login Route*

| // Login route router.post('/login', async (req, res) => {  try {  // Extract username and password from request body  const { username, password } = req.body;   // Find the user by username  const user = await Person.findOne({ username });   // If user does not exist or password does not match, return error  if (!user || !(await user.comparePassword(password))) {  return res.status(401).json({ error: 'Invalid username or password' });  }   const payload = { id: user.id, email: user.email }   // Generate JWT token  const token = generateToken(payload);   // Send token in response  res.json({ token });  } catch (err) {  console.error(err);  res.status(500).json({ error: 'Internal Server Error' });  } }); |
| --- |

### 

### 

### *Profile Route*

| router.get('/profile', jwtAuthMiddleware, async (req, res) => {  try {  // Extract user id from decoded token  const userId = req.user.id;   // Find the user by id  const user = await Person.findById(userId);   // If user does not exist, return error  if (!user) {  return res.status(404).json({ error: 'User not found' });  }   // Send user profile as JSON response  res.json(user);  } catch (err) {  console.error(err);  res.status(500).json({ error: 'Internal Server Error' });  } }); |
| --- |

Important Points:

When the server receives this request, it parses the Authorization header to extract the JWT token.

| const authorizationHeader = 'Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c'; const token = authorizationHeader.split(' ')[1]; console.log(token); // Output: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV\_adQssw5c |
| --- |

The split(' ') method separates the header value into an array of substrings using the space (' ') character as the delimiter. The [1] index accesses the second element of this array, which is the JWT token.

We can Add one more line of check in the jwtAuthMiddleware

| const authHeader = req.headers.authorization;  if (!authHeader) return res.status(401).json({ error: 'Token required' }); |
| --- |

### *Expire Time-based Token*

