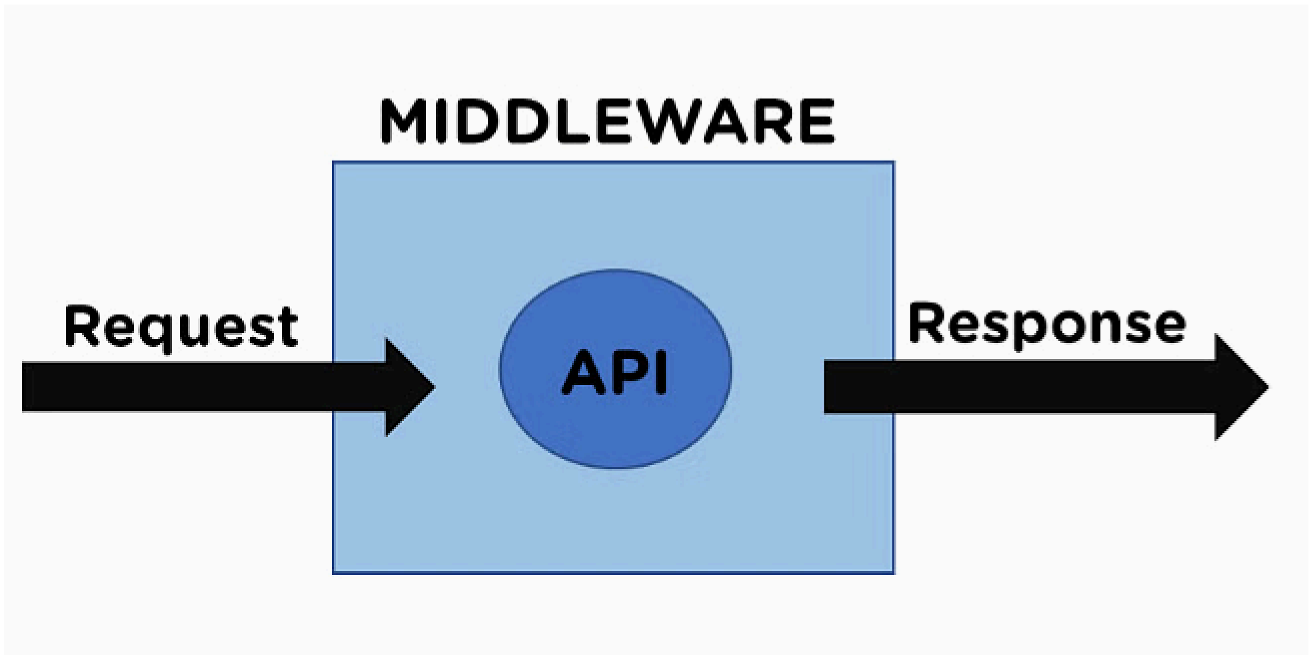
DAY 8

- *Middleware*

Imagine you're at a restaurant, and you've placed an order for your favorite dish. Now, before that dish reaches your table, it goes through several stages in the kitchen. Each stage involves different tasks, like chopping vegetables, cooking, and adding spices. Middleware is a bit like these stages in the kitchen—it's something that happens in between your request and the final response in a web application.

Now, let's apply this idea to a web application, like the "Node Hotel" system:

1. **Request Phase:**
    - You (the client) make a request to the Node Hotel system. It could be asking for the menu, submitting a reservation, or anything else.
2. **Middleware Phase:**
    - Middleware is like the behind-the-scenes process in the kitchen. It's a series of functions that your request goes through before it reaches the final destination.
3. **Final Response Phase:**
    - After passing through the middleware, your request gets processed, and the system sends back a response. It could be the menu you requested or confirmation of your reservation.



Example in Node.js:

In the context of Node.js, imagine you want to log every request made to your "Node Hotel" application. You could use middleware for this.

```
// Middleware Function
const logRequest = (req, res, next) => {
    console.log(`[${new Date().toLocaleString()}] Request made to: ${req.originalUrl}`);
    next(); // Move on to the next phase
};

// Using Middleware in Express
const express = require('express');
const app = express();

// Apply Middleware to all Routes
app.use(logRequest);

// Define Routes
app.get('/', (req, res) => {
    res.send('Welcome to Node Hotel!');
});

app.get('/menu', (req, res) => {
    res.send('Our delicious menu is coming right up!');
});

// Start the Server
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

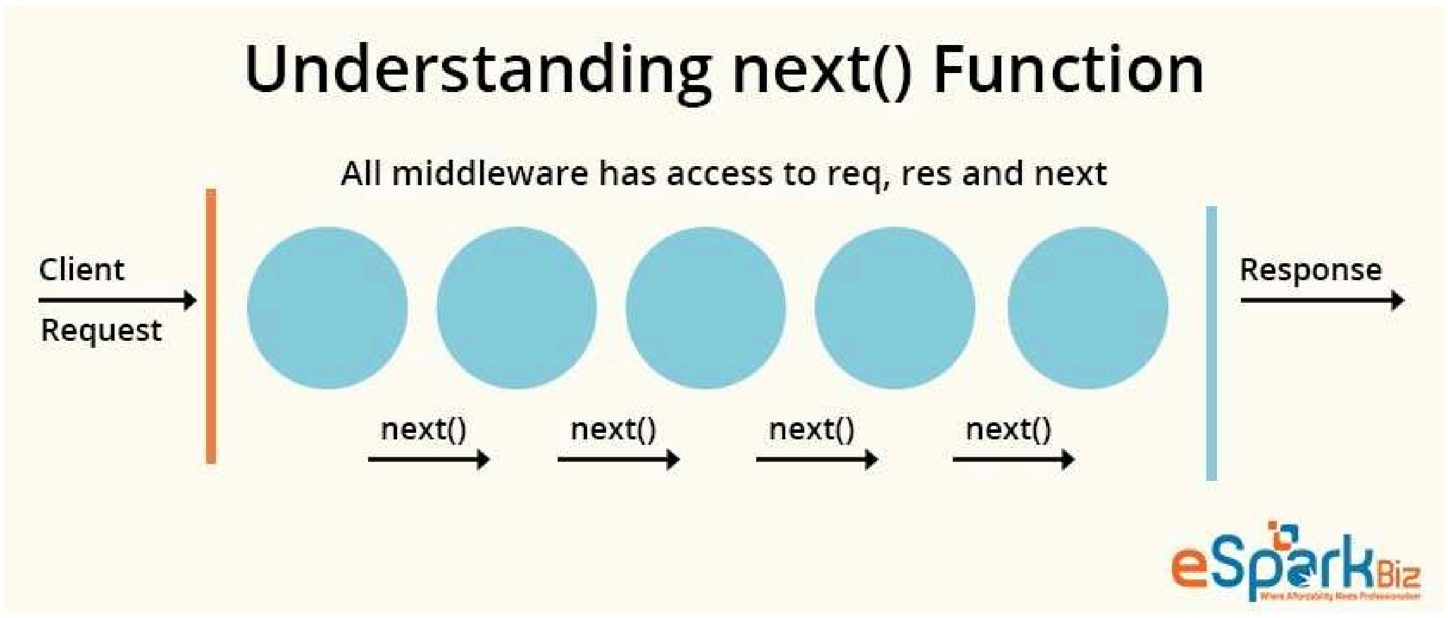
In this example, `logRequest` is our middleware. It logs the time and the requested URL for every incoming request. The `app.use(logRequest)` line tells Express to use this middleware for all routes.

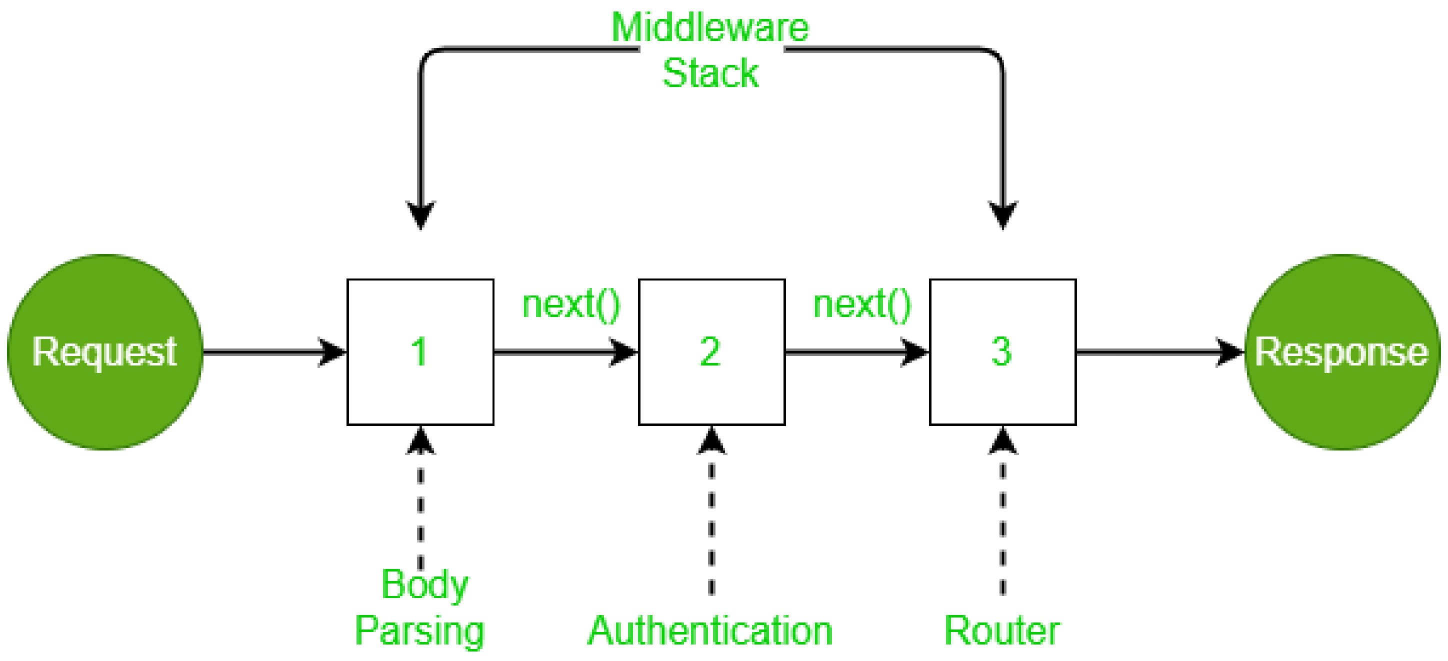
So, when you access any route (like `/` or `/menu`), the middleware runs first, logs the request, and then the route-specific code executes.

In summary, middleware is like a series of tasks that happen behind the scenes in a web application. **It's a way to add extra functionality to your application's request-response cycle**, such as **logging**, **authentication checks**, or **modifying request data**, before it reaches its final destination.

Why do we use the next() function in the middleware function

In Express.js, the `next()` function is a callback that *signals to Express* that the current middleware function has completed its processing and that it's time to move on to the next middleware function or route handler in the chain.


Understanding next() Function
All middleware has access to req, res and next
Client
Request
Response
next()
next()
next()
next()
eSparkBiz


Middleware Stack
Request
1
next()
2
next()
3
Response
Body Parsing
Authentication
Router

- *Authentication & Authorization*

Like Aim: 700
Comment Aim: 400
Please Like this Video Before watching

Tag me on Linkedin for Notes – show Demo How you can post

Imagine you're the manager of the "Node Hotel" application, and you want to ensure that only authorized staff members can access certain features. This is where authentication comes in.

1. Verifying Identity (Authentication):

- **Scenario:** When a staff member, let's say a chef, wants to log in to the Node Hotel system, they need to prove that they are indeed the chef they claim to be.
- **In Practice:** In Node.js, authentication involves checking the chef's credentials, like a username and password, to make sure they match what's on record. It's like asking the chef to enter a secret code (password) and confirming that it's correct.

2. Access Control (Authorization):

Now, let's add a layer of authorization based on the roles of the staff members.

- **Scenario:** Once the chef has proven their identity, you, as the manager, want to control what they can and cannot do. For instance, chefs should be able to update the menu items, but maybe not manage staff salaries.
- **In Practice:** In Node.js, after authenticating the chef, you'll use authorization to decide what parts of the system they have access to. It's like giving the chef a key card (authorization) that lets them into the kitchen but not into the manager's office.

Implementation in Node.js:

1. **Authentication Middleware:**
   - In your Node.js application, you might use middleware like Passport to handle the authentication process.
   - Passport helps verify the identity of the chef based on their provided credentials.
2. **User Roles and Permissions:**
   - You'll define roles for staff members (e.g., chef, waiter, manager).
   - Authorization middleware will check the role of the authenticated user and grant access accordingly.

3. **Secure Endpoints:**
    - You'll protect certain routes (like updating menu items) with authentication checks.
    - Only authenticated and authorized users (like chefs) will be allowed to access these routes.

In the Hotel Context:

- **Authentication:** When Chef John logs in, the system checks if the provided username and password match what's on record for Chef John.
- **Authorization:** Once authenticated, Chef John is authorized to modify menu items but may not have permission to change other critical settings.

In simple terms, authentication in Node.js for your hotel application ensures that each staff member is who they say they are, and authorization determines what they're allowed to do once their identity is confirmed.

It's like having a secure system where only the right people get access to the right areas of your hotel management application.

In general, authentication is applied before authorization in the security process. Here's the typical sequence:

1. **Authentication:**
    - The first step is to verify the identity of the user or system entity attempting to access a resource or perform an action. This involves checking credentials such as usernames and passwords or using other authentication methods like tokens, API keys, or certificates.
2. **Authorization:**
    - Once the identity is verified through authentication, the system moves on to authorization. Authorization determines what actions or resources the authenticated user or entity is allowed to access based on their permissions, roles, or other access control mechanisms.

The reason for this order is straightforward: before you can determine what someone is allowed to do (authorization), you need to know who they are (authentication). Authentication establishes the identity, and authorization defines the permissions associated with that identity.

In the context of web applications, middleware for authentication is typically applied first in the request-response cycle to verify the user's identity. If authentication is successful, the request proceeds to authorization middleware to determine what the authenticated user is allowed to do.

It's important to note that while authentication and authorization are often discussed as distinct steps, they work together as essential components of a security strategy to control access to resources and protect against unauthorized actions.

- Now we will implement Authentication as a middleware Function. So that, Routes will be authenticated before reaching out to the server.
- Implementing authentication as a middleware function is a common and effective approach.

- *Passport.js*

Passport.js is a popular **authentication middleware for Node.js**. Authentication is the process of verifying the identity of a user, typically through a username and password, before granting access to certain resources or features on a website or application.

Think of Passport.js as a helpful tool that makes it easier for developers to handle user authentication in their Node.js applications. It simplifies the process of authenticating users by **providing** a set of pre-built strategies for different authentication methods, such as username and password, social media logins (like Facebook or Google), and more.

Here's a breakdown of some key concepts in Passport.js:

1. **Middleware:** In the context of web development, *middleware is software that sits between the application and the server.* Passport.js acts as middleware, intercepting requests and adding authentication-related functionality to them.
2. **Strategy:** Passport.js uses the concept of strategies for handling different authentication methods. A strategy is a way of authenticating users. Passport.js comes with various built-in strategies, and you can also create custom strategies to support specific authentication providers.
3. **Serialize and Deserialize:** Passport.js provides methods for serializing and deserializing user data. Serialization is the process of converting user data into a format that can be stored, usually as a unique identifier. Deserialization is the reverse process of converting that unique identifier back into user data. These processes are essential for managing user sessions.

- Install Passport

To use Passport.js in a Node.js application, you need to **install the passport package** along with the **authentication strategies** you intend to use.

For this course, we are using Local strategies authentication (username and password).

you would typically install `passport-local`

```
npm install passport passport-local
```

Once you've installed these packages, you can set up and configure Passport.js in your application.

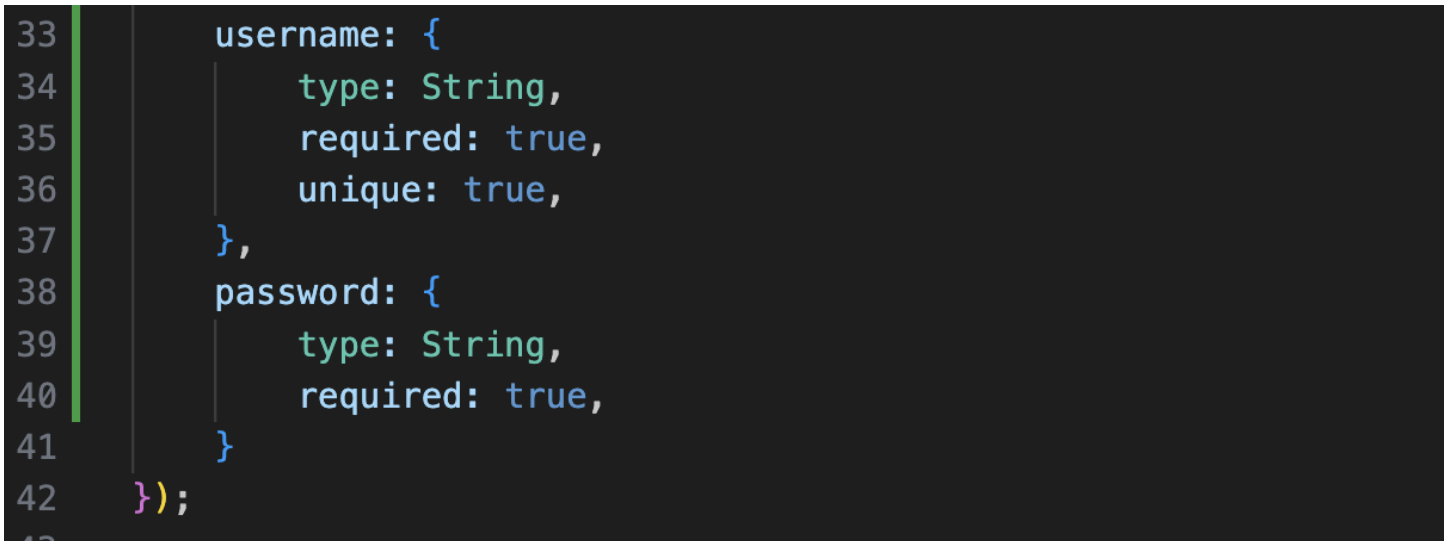
```
const express = require('express');
const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const app = express();
// Initialize Passport
app.use(passport.initialize());
```

- Passport Local Strategy

- The Passport Local Strategy is a part of the Passport.js authentication middleware for Node.js. It's specifically designed for handling username and password-based authentication.
- The Passport Local Strategy, by default, expects to extract the username and password from the request body. It is a common practice for username and password-based authentication systems to send the credentials as part of the request body, especially in login forms.

- Add username & password

- Now we have to add username and password in the person schema



```
    username: {
        type: String,
        required: true,
        unique: true,
    },
    password: {
        type: String,
        required: true,
    }
});
```

- Configure the Local Strategy

- Define and configure the Local Strategy using `passport-local`.
- You need to provide a verification function that checks the provided username and password.

```
passport.use(new LocalStrategy(
  async (username, password, done) => {
    // Your authentication logic here
  }
));
```

- In the Local Strategy's verification function, you typically query your database to find the user with the provided username. You then compare the provided password with the stored password.
- In the context of `LocalStrategy`, Passport.js expects the verification function to have the following signature:

```
function(username, password, done)
```

- The done callback should always be the last parameter, and it's essential to maintain this order for Passport.js to work correctly. If you change the order of parameters, you risk breaking the expected behavior of Passport.js.

```
passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = (user.password === password ? true : false);
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));
```

- In the context of Passport.js, `done` is a callback function that is provided by Passport to signal the completion of an authentication attempt. It is used to indicate whether the authentication was successful, and if so, to provide information about the authenticated user.
- The `done` function takes three parameters: `done(error, user, info)`.
- If the authentication is successful, you call `done(null, user)` where `user` is an object representing the authenticated user.
- If the authentication fails, you call `done(null, false, { message: 'some message' })`. The second parameter (`false`) indicates that authentication failed, and the third parameter is an optional info object that can be used to provide additional details about the failure.

- Passport Authenticate

- Once you have configured Passport Local Strategy, the next steps typically involve integrating it into your application.
- In route, we should use `passport.authenticate()` to initiate the authentication process.
- To authenticate any routes, we need to pass this as an middleware

Let’s suppose we want to Authenticate this Route

```
app.get('/', function (req, res) {
    res.send('Welcome to our Hotel');
})
```

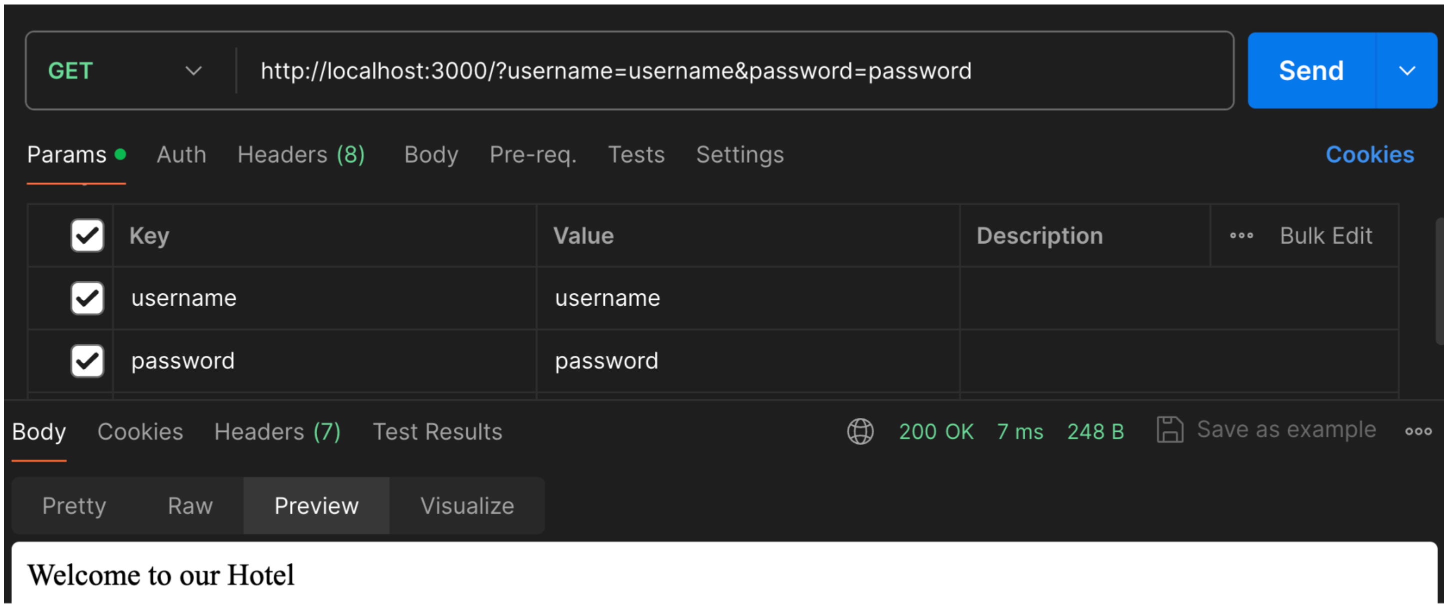
- We have to initialize the passport

```
app.use(passport.initialize());
app.get('/', passport.authenticate('local', { session: false }), function (req, res) {
    res.send('Welcome to our Hotel');
})
```

Or we can also write like this

```
app.use(passport.initialize());
const localAuthMiddleware = passport.authenticate('local', { session: false });

app.get('/', localAuthMiddleware, function (req, res) {
    res.send('Welcome to our Hotel');
})
```

- Now, we can test the ‘ / ‘routes it needs parameter



http://localhost:3000/?username=username&password=password

	Key	Value	Description
✓	username	username	
✓	password	password	

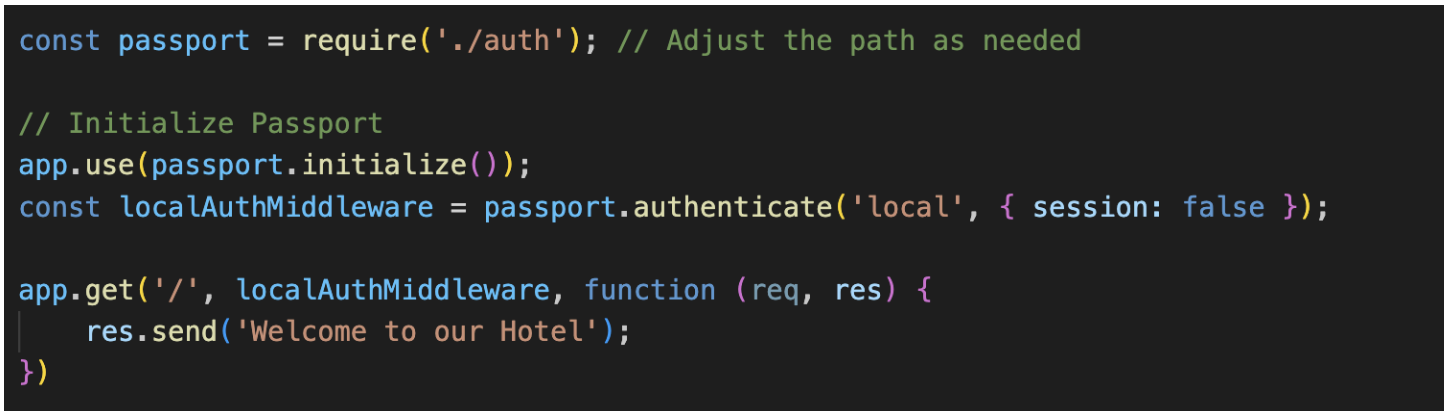
200 OK 7 ms 248 B

Welcome to our Hotel

- Same way, we can also pass authenticate /person routes.

- Passport Separate File

- Now, rather than all the passport codes in a server file. We can separate it into another file name auth.js
- And in the server.js file, we will import the passport

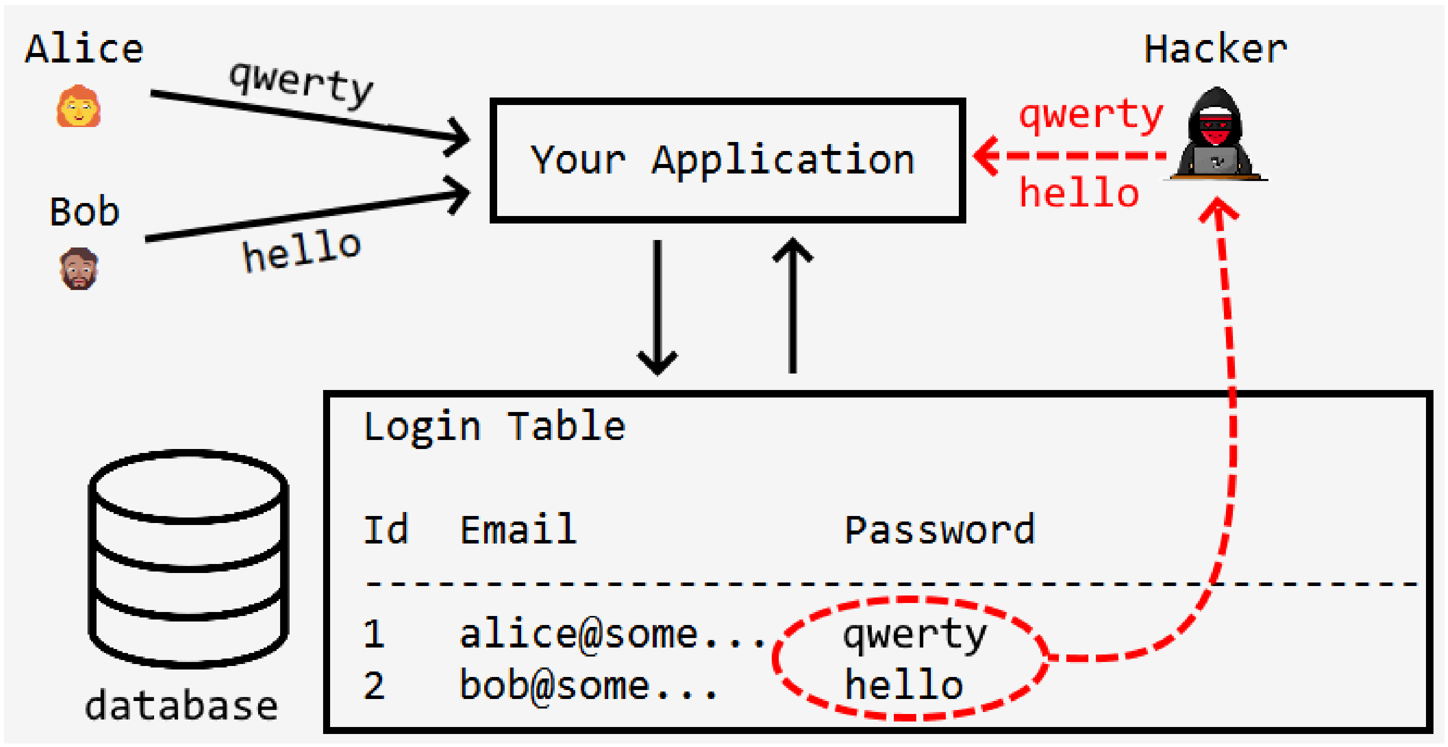


```
const passport = require('./auth'); // Adjust the path as needed

// Initialize Passport
app.use(passport.initialize());
const localAuthMiddleware = passport.authenticate('local', { session: false });

app.get('/', localAuthMiddleware, function (req, res) {
    res.send('Welcome to our Hotel');
})
```

```javascript
// sets up Passport with a local authentication strategy, using a Person model
for user data. - Auth.js file

const passport = require('passport');
const LocalStrategy = require('passport-local').Strategy;
const Person = require('./models/Person'); // Adjust the path as needed

passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = user.password === password ? true : false;
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));

module.exports = passport; // Export configured passport
```

- *Store Plain Password*

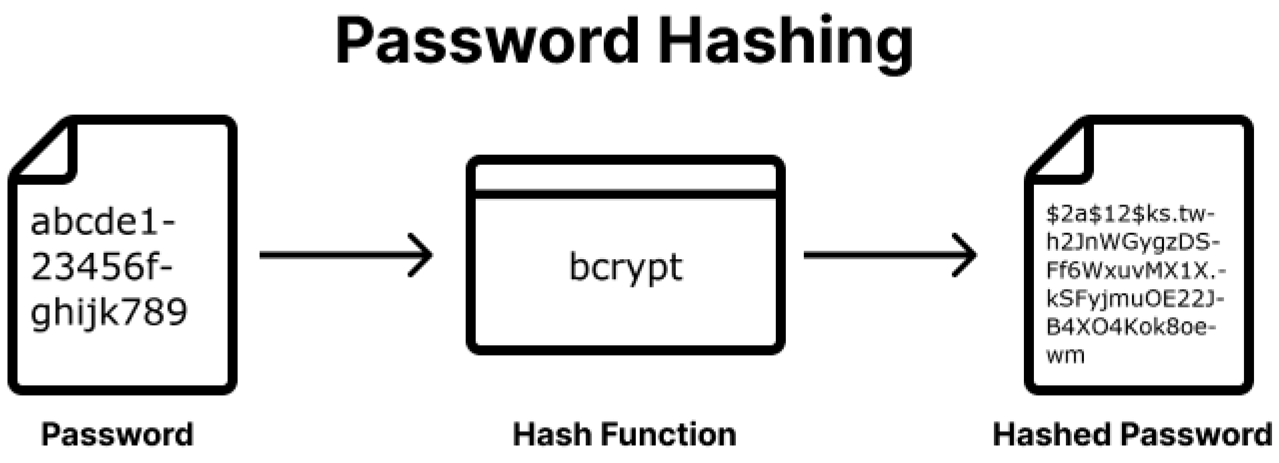
- Storing plain passwords is not a secure practice. To enhance security, it's highly recommended to hash and salt passwords before storing them.

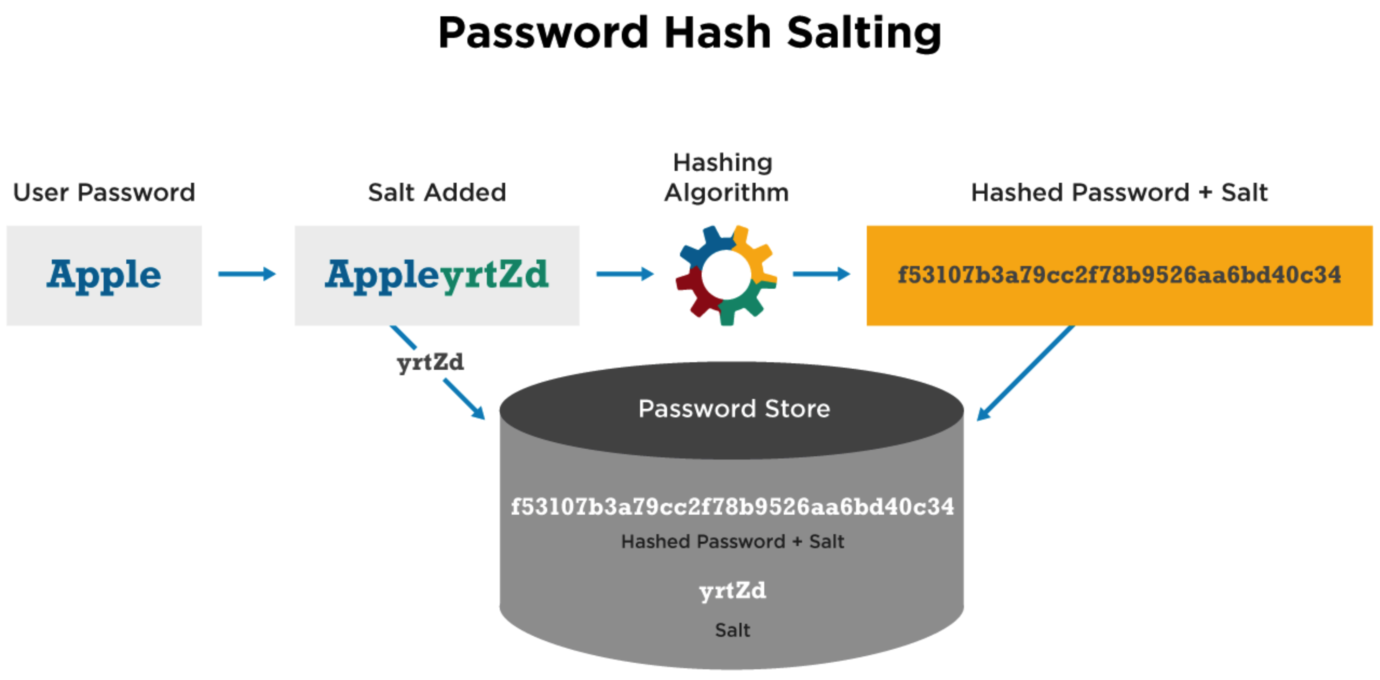


- You can use the `bcrypt` library for password hashing in your Node.js application.

```
npm install bcrypt
```

- *bcrypt.js*





- Now we have to update our person model to store hashed passwords. Modify the registration logic to hash the password before saving it to the database.
- Because the end user didn't know about hashing, we have to internally maintain it. Like we are saving the hashed password before saving it into the database
- We are using a Mongoose middleware hook to perform an action before saving a document to the database. Specifically, it's using the `pre` middleware to execute a function before the `save` operation.

```
personSchema.pre('save', async function(next) {
    const person = this;

    // Hash the password only if it has been modified (or is new)
    if (!person.isModified('password')) return next();

    try {
        // Generate a salt
        const salt = await bcrypt.genSalt(10);

        // Hash the password with the salt
        const hashedPassword = await bcrypt.hash(person.password, salt);

        // Override the plain password with the hashed one
        person.password = hashedPassword;
        next();
    } catch (error) {
        return next(error);
    }
});
```

- The `pre('save', ...)` middleware is triggered before the `save` operation on a Mongoose model instance.
- Inside the middleware function, it checks if the password field has been modified (or if it's a new document). If not, it skips the hashing process.
- If the password has been modified, it generates a new salt using `bcrypt.genSalt` and then hashes the password using `bcrypt.hash`.
- The original plain text password in the `person` document is then replaced with the hashed password.
- The `next()` function is called to proceed with the save operation.

- The line `const salt = await bcrypt.genSalt(10);` is responsible for generating a salt, which is a random string of characters used as an additional input to the password hashing function. Salting is a crucial step in password hashing to prevent attackers from using precomputed tables (rainbow tables) to quickly look up the hash value of a password.

- `bcrypt.genSalt(rounds)`: This function generates a salt using the specified number of "rounds." The `rounds` parameter indicates the complexity of the hashing algorithm. The higher the number of rounds, the more secure the salt, but it also increases the computational cost.

```
if (!person.isModified('password')) return next();
```

This line is a conditional check that prevents unnecessary rehashing of the password when the document is being saved.

- `person.isModified('password')`: This method is provided by Mongoose and returns `true` if the specified field (`'password'` in this case) has been modified. It returns `false` if the field hasn't been modified.
- `return next();`: If the password field has not been modified, the function immediately returns, skipping the rest of the middleware. This is because there's no need to rehash the password if it hasn't changed.

- How `bcrypt` works

When you use `bcrypt` to hash a password, the library internally stores the salt as part of the resulting hashed password. This means that you don't need to separately store the salt in your database; it is included in the hashed password itself.

Here's a simplified explanation of how it works:

1. **Hashing a Password:**

When you hash a password using `bcrypt.hash`, the library generates a random salt, hashes the password along with the salt, and produces a hashed password that incorporates both the salt and the hashed value.

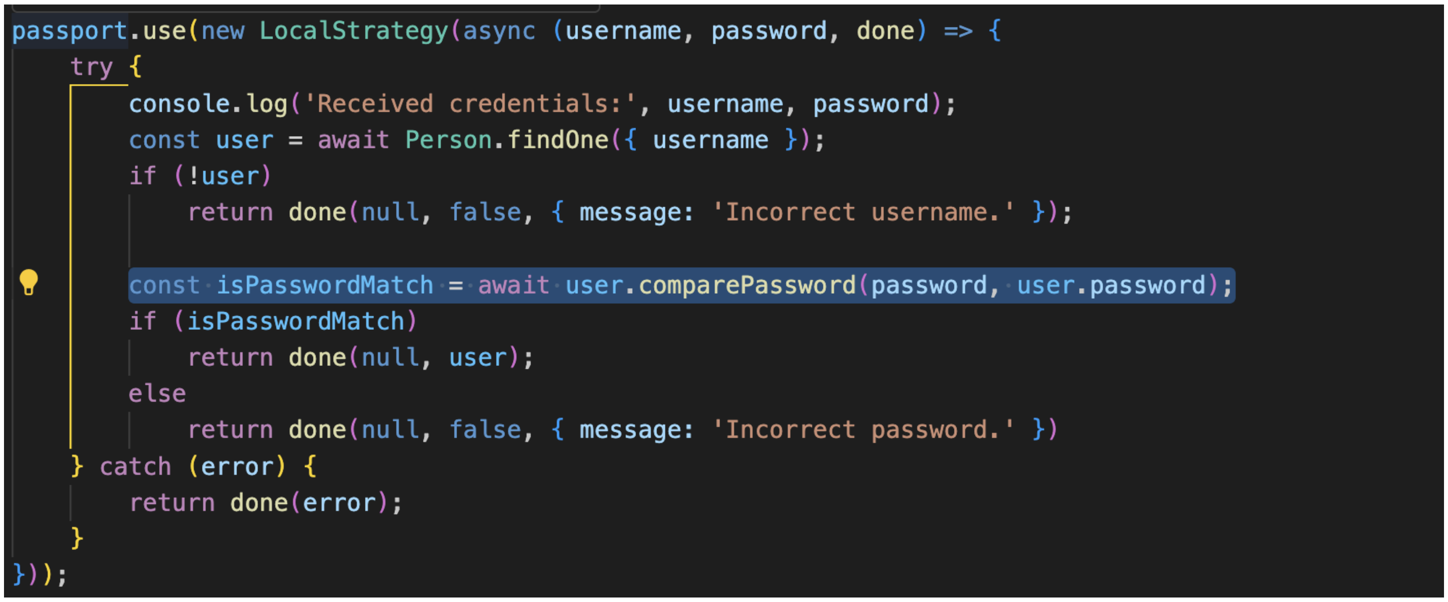
```
const salt = await bcrypt.genSalt(10);
const hashedPassword = await bcrypt.hash('userPassword', salt);
```

The `hashedPassword` now contains both the hashed password and the salt.

- Modify Auth code

We have to also modify the password-matching logic in the passport auth file.

- Let’s create a **comparePassword** named function which compares or check the password.



```
passport.use(new LocalStrategy(async (username, password, done) => {
    try {
        console.log('Received credentials:', username, password);
        const user = await Person.findOne({ username });
        if (!user)
            return done(null, false, { message: 'Incorrect username.' });

        const isPasswordMatch = await user.comparePassword(password, user.password);
        if (isPasswordMatch)
            return done(null, user);
        else
            return done(null, false, { message: 'Incorrect password.' })
    } catch (error) {
        return done(error);
    }
}));
```

- Compare function

We also have to write the compare function

```
// Define the comparePassword method
personSchema.methods.comparePassword = async function(candidatePassword) {
    try {
        // Use bcrypt to compare the provided password with the hashed password
        const isMatch = await bcrypt.compare(candidatePassword, this.password);
        return isMatch;
    } catch (error) {
        throw error;
    }
};
```

Verifying a Password

When you later want to verify a user's entered password during login, you use `bcrypt.compare`. This function internally extracts the salt from the stored hashed password and uses it to hash the entered password for comparison.

```
const isMatch = await bcrypt.compare('enteredPassword', storedHashedPassword);
```

The `compare` function automatically extracts the salt from `storedHashedPassword` and uses it to hash the entered password. It then compares the resulting hash with the stored hash. If they match, it indicates that the entered password is correct.

DAY 9

Sessions or Tokens

Sessions are a way to maintain user state and authenticate users in web applications.

1. User Authentication

When you visit a website or use a web application, you must log in to access certain features or personalized content. Sessions or tokens are used to authenticate users, proving that they are who they claim to be. This ensures that only authorized users can access restricted areas or perform certain actions.

2. Maintaining User State

Web applications often need to remember user information as they navigate through different pages or interact with the app. Sessions or tokens help maintain this user state. For example, if you add items to your shopping cart on an e-commerce website, the website needs to remember those items as you browse other pages. Sessions or tokens store this information so that it can be accessed later.

3. Security

Sessions and tokens play a crucial role in securing web applications. They help prevent unauthorized access by verifying the identity of users before granting them access to sensitive data or functionality. Additionally, they can be used to protect against common security threats such as session hijacking or cross-site request forgery (CSRF) attacks.

4. Personalization and Customization

Sessions or tokens allow web applications to provide personalized experiences to users. By storing information about users' preferences, settings, or past interactions, the application can tailor the content or functionality to better meet their needs. This enhances user satisfaction and engagement with the application.

5. Scalability

Sessions and tokens are designed to scale with the size and complexity of web applications. They provide a flexible and efficient way to manage user authentication and state, even as the number of users or requests increases. This scalability is essential for ensuring that web applications can handle growing traffic and user demands without sacrificing performance.

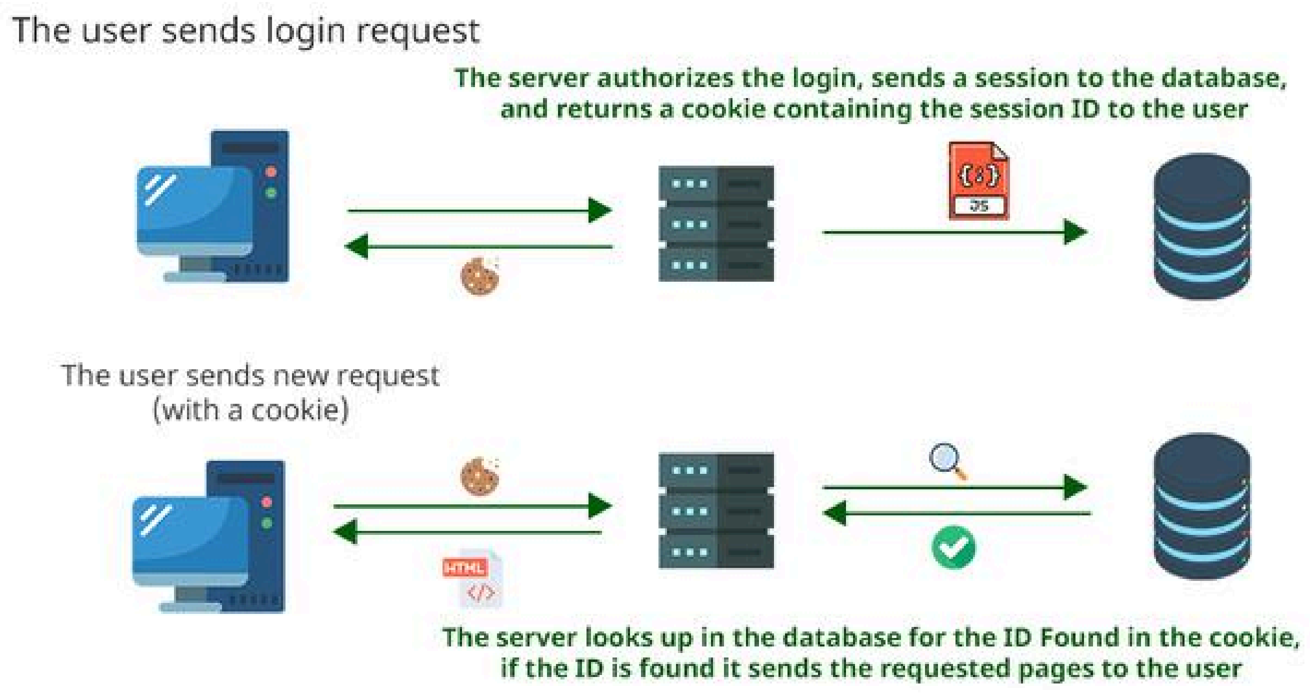
In summary, sessions and tokens are essential components of web development, enabling user authentication, maintaining state, ensuring security, personalizing experiences, and supporting scalability. They form the foundation of secure and user-friendly web applications that can provide personalized and seamless experiences to users.

Sessions based Authentication

- A session is a way or a small file, most likely in JSON format, that stores information about the user, such as a unique ID, time of login expirations, and so on. It is generated and stored on the server so that the server can keep track of the user requests.

Working

1. The user sends a login request to the server.
2. The server authenticates the login request, sends a session to the database, and returns a cookie containing the session ID to the user.
3. Now, the user sends new requests (with a cookie).
4. The server checks in the database for the ID found in the cookie, if the ID is found it sends the requested pages to the user.


The user sends login request
The server authorizes the login, sends a session to the database, and returns a cookie containing the session ID to the user
JS
The user sends new request (with a cookie)
HTML
</>
The server looks up in the database for the ID Found in the cookie, if the ID is found it sends the requested pages to the user

Cookies

A cookie is a **small piece of data** **that a website stores** on a **user's computer or device**. Cookies are commonly used by websites to remember information about the user's browsing activity, preferences, and interactions with the site. Here's a beginner-friendly explanation of cookies

How Cookies Work

- **Creation**: When you visit a website for the first time, the website may send a small text file (the cookie) to your browser.
- **Storage**: Your browser stores the cookie on your computer or device. Each cookie is associated with a specific website and contains information such as a unique identifier and any data the website wants to remember about you.
- **Sending with Requests**: When you revisit the website or navigate to different pages on the same site, your browser automatically includes the cookie in the HTTP requests it sends to the website's server.
- **Server Processing**: The website's server receives the cookie along with the request. It can then read the cookie to retrieve the stored information about you.
- **Usage**: Websites use cookies for various purposes, such as remembering your login status, storing your preferences, tracking your activities for analytics purposes, and personalizing your browsing experience.

Types of Cookies

- **Session Cookies**: These cookies are temporary and are deleted when you close your browser. They are often used for maintaining your login state during a browsing session.
- **Persistent Cookies**: These cookies are stored on your device for a specified duration, even after you close your browser. They can be used for purposes such as remembering your preferences or login information across multiple visits to a website.

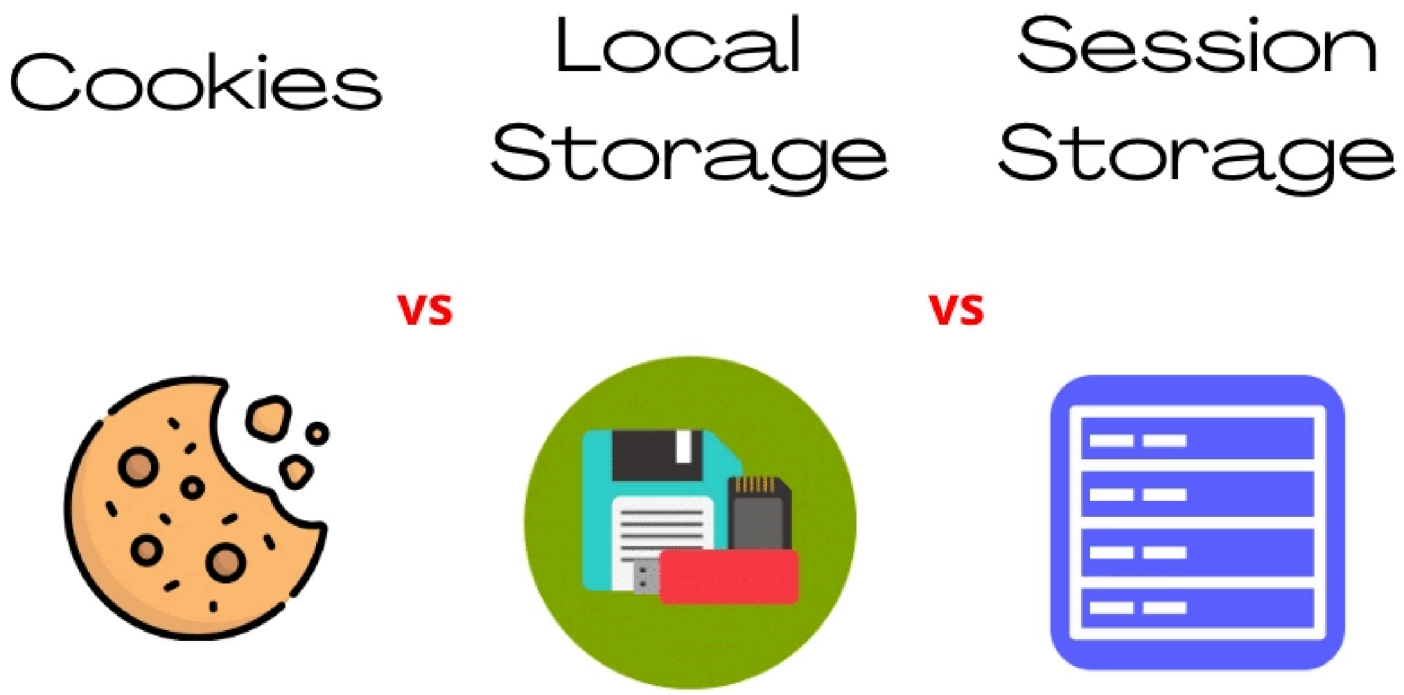
Privacy and Security

Cookies can raise privacy concerns because they can be used to track users' browsing behavior across different websites. However, cookies cannot execute code or transmit viruses, and they can only store information that the website itself provides.

*In summary, **cookies are small text files** stored on your computer by websites you visit. They help websites remember information about you and enhance your browsing experience by personalizing content, maintaining login status, and tracking preferences. While cookies are an essential part of web functionality, they also raise privacy considerations, and users can manage their cookie settings in their web browsers.*

Client Storage option

It's not mandatory that the cookies received from the server will be stored in the "Cookies" storage of your browser. The browser provides multiple storage options, each serving different purposes and offering various capabilities:



1. **Cookies Storage:** Cookies received from the server are typically stored in the "Cookies" storage of your browser. These cookies can include session cookies, which are deleted when you close your browser, and persistent cookies, which are stored for a longer period.
2. **Local Storage:** Local storage is a mechanism that allows web applications to store data locally in the browser. Unlike cookies, data stored in local storage is not automatically sent to the server with every request. This storage option is often used for caching data, storing user preferences, or implementing client-side features.

3. **Session Storage:** Session storage is similar to local storage but is scoped to a particular browsing session. Data stored in session storage is cleared when you close the browser or tab. It's commonly used for temporary data storage or for maintaining a state within a single browsing session.

Important Points

- While cookies are commonly used for storing session identifiers (session IDs) or authentication tokens received from the server, web applications can choose to store this information in local storage or session storage instead.
- The choice of storage mechanism depends on factors such as security requirements, application architecture, and use case. Cookies offer automatic inclusion in HTTP requests and can be more secure if configured properly, but local storage and session storage provide more control over data management on the client side.
- While cookies are a common way to store data received from the server, web applications have the flexibility to choose other storage options such as local storage or session storage based on their requirements.

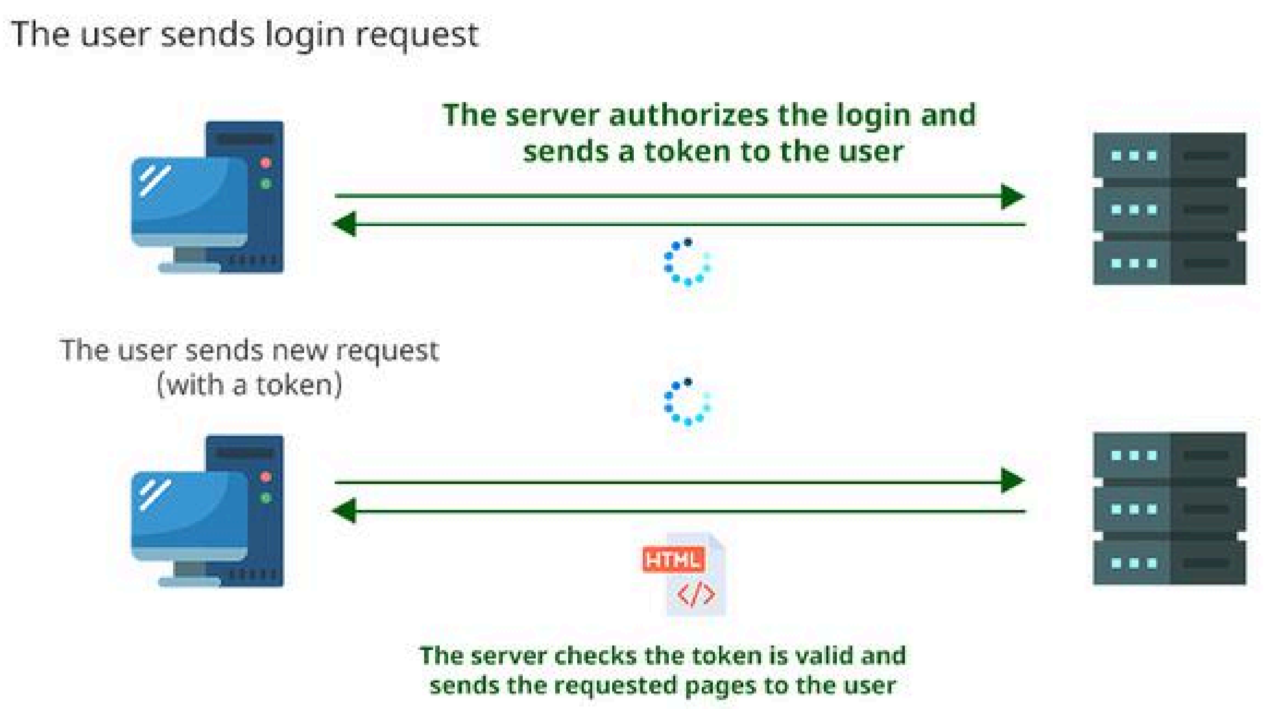
Token based Authentication

- A token is an authorization file that cannot be tampered with. It is generated by the server using a secret key, sent to and stored by the user in their local storage. Like in the case of cookies, the user sends this token to the server with every new request, so that the server can verify its signature and authorize the requests.

- Working

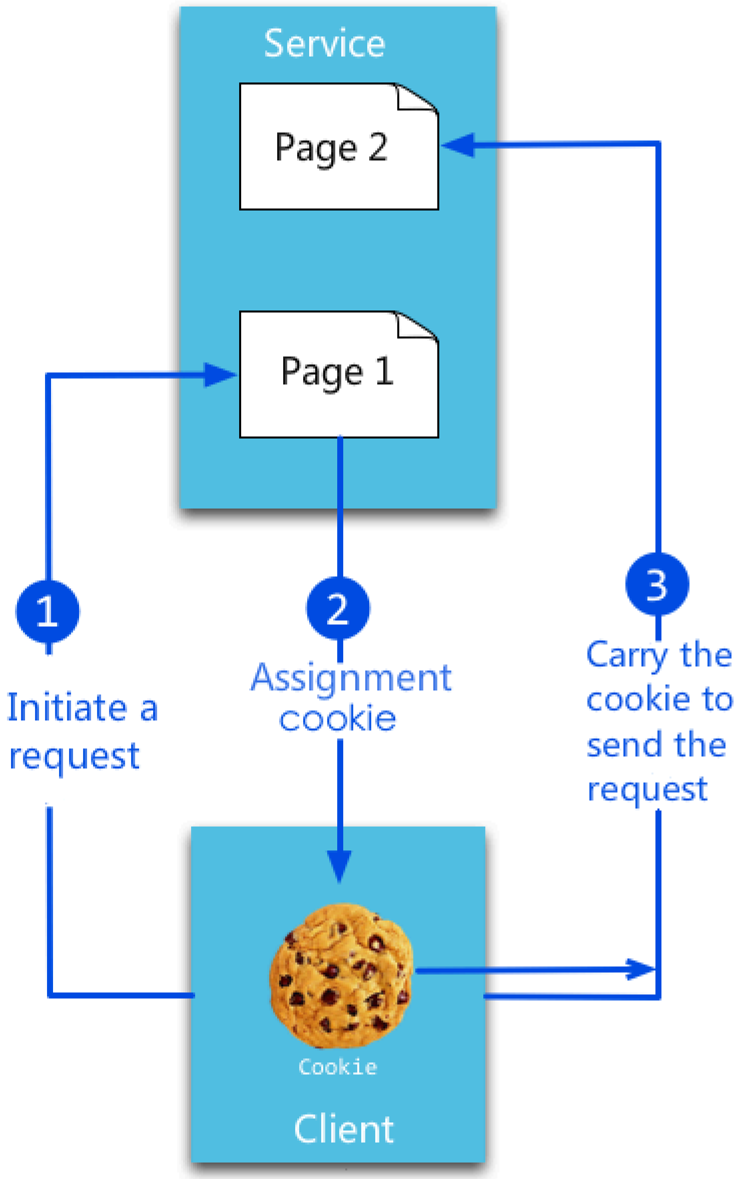
- The user sends a login request to the server.
- The server authorizes the login and sends a token to the user.
- Now, the user sends a new request(with a token).
- The server checks whether the token is valid or not, if the token is valid it sends the requested pages to the user.

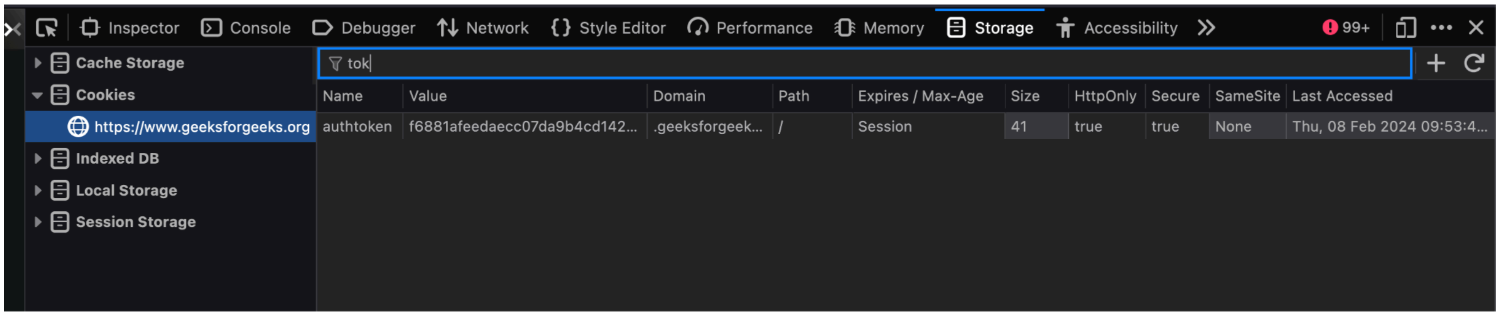
*Note- Those are **not authentication files,** they are **authorization ones**. While receiving a token, the server does **not** look up who the user is, it simply authorizes the user's requests relying on the validity of the token.*


The user sends login request
The server authorizes the login and sends a token to the user
The user sends new request (with a token)
HTML
The server checks the token is valid and sends the requested pages to the user

Difference between Session & Token

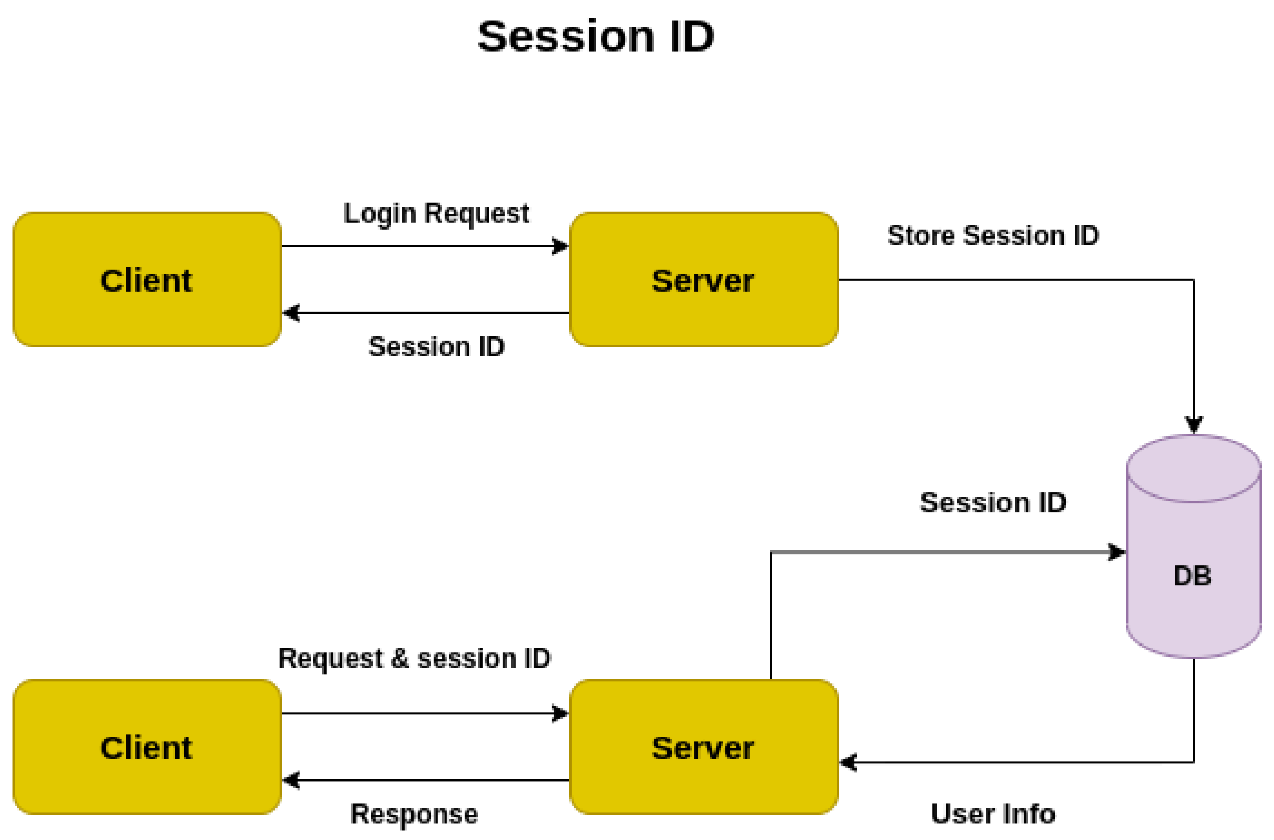
	Criteria	**Session authentication method**	**Token-based authentication method**
1	Which side of the connection stores the authentication details	Server	User
2	What the user sends to the server to have their requests authorized	A cookie	The token itself
3	What does the server do to authorize users' requests	Looking up in its databases to find the right session thanks to the ID the user sends with a cookie	Decrypting the user's token and verifying its signature
4	Can the server admins perform securities operations like logging users out, changing their details, etc	Yes, because the session is stored on the server	No, because the token is stored on the user's machine



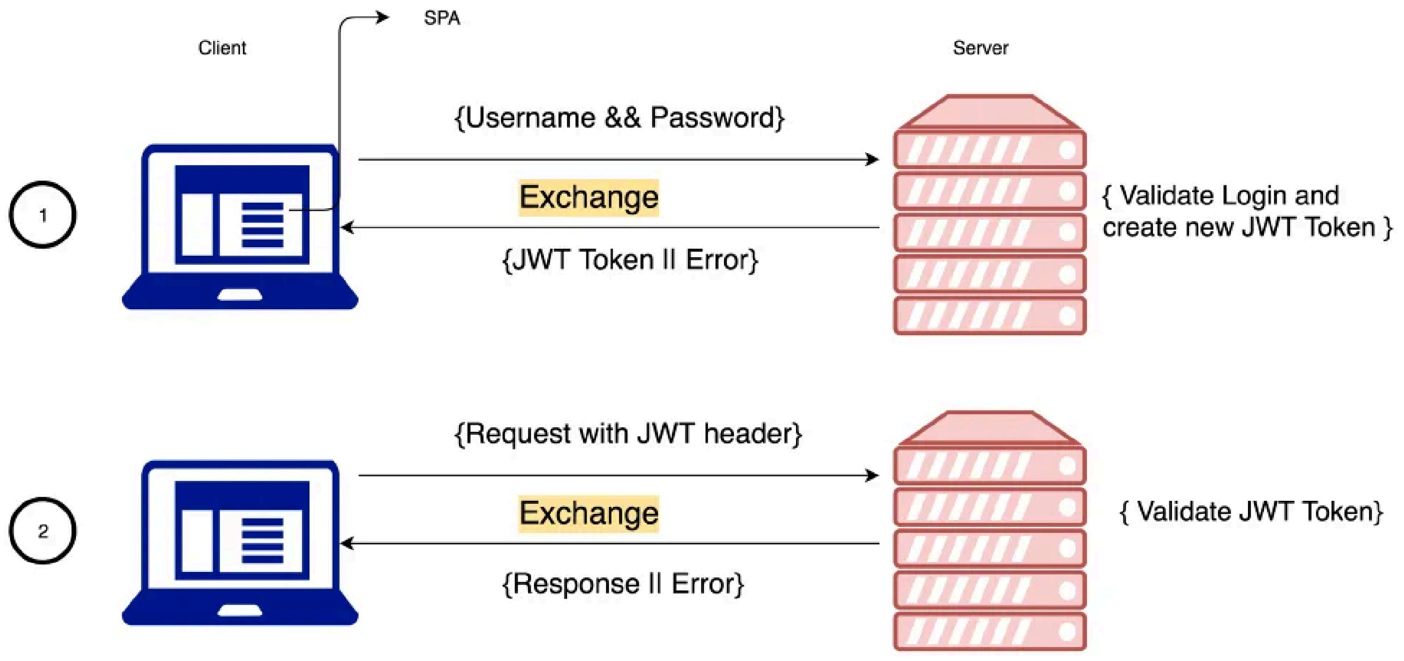


Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite	Last Accessed
authtoken	f6881afeedaecc07da9b4cd142...	.geeksforgeek...	/	Session	41	true	true	None	Thu, 08 Feb 2024 09:53:4...

How Session works



How Token works



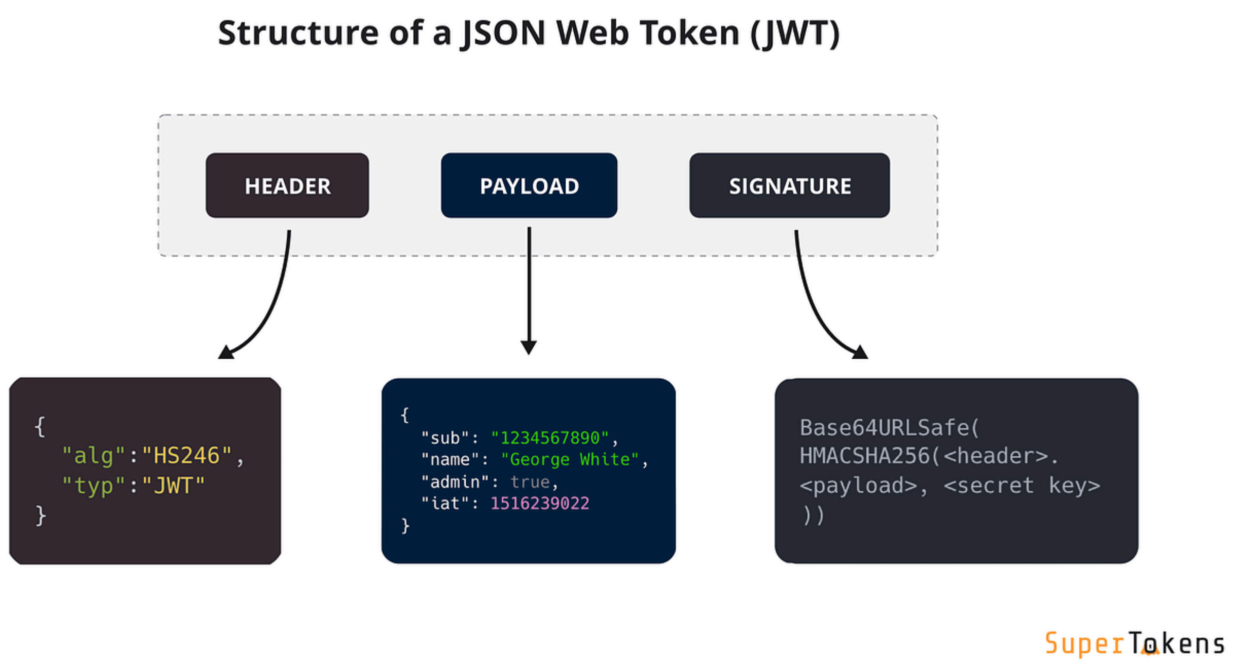
DAY 10

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.



```
{
  "alg":"HS246",
  "typ":"JWT"
}
```

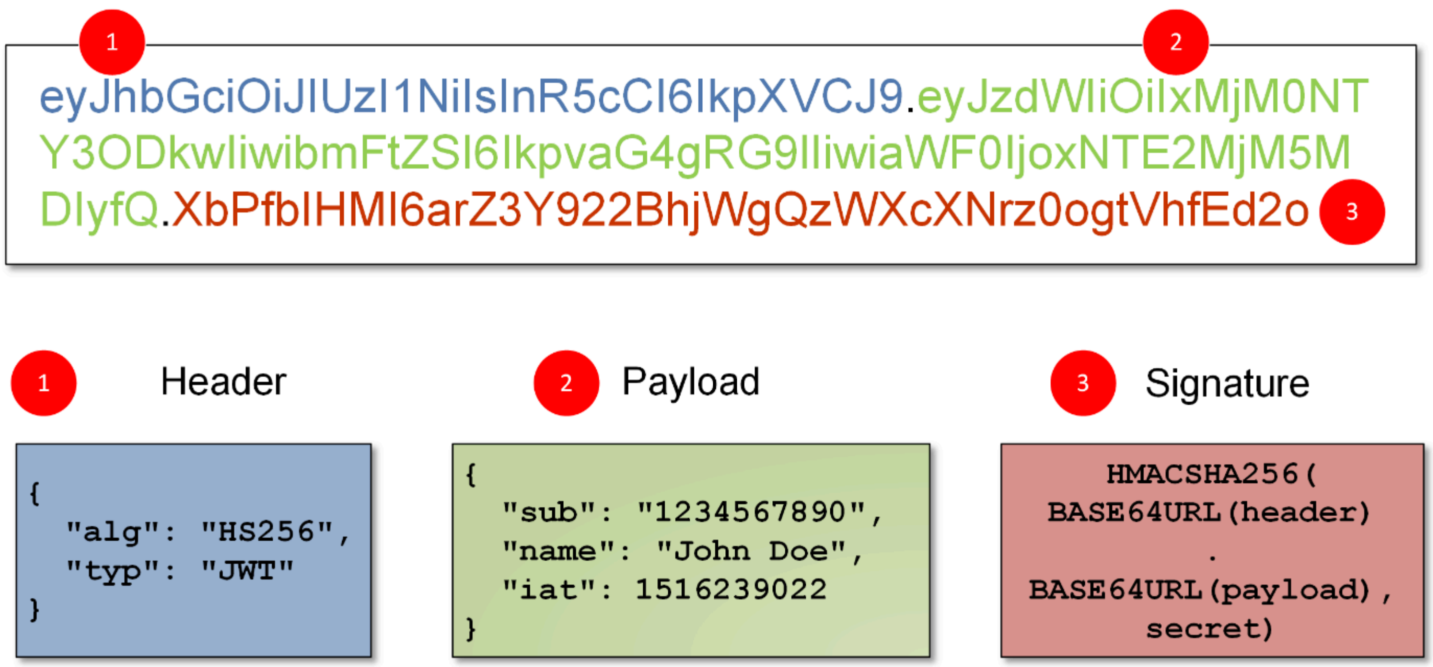
```
{
  "sub": "1234567890",
  "name": "George White",
  "admin": true,
  "iat": 1516239022
}
```

```
Base64URLSafe(
HMACSHA256(<header>.
<payload>, <secret key>
))
```

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.



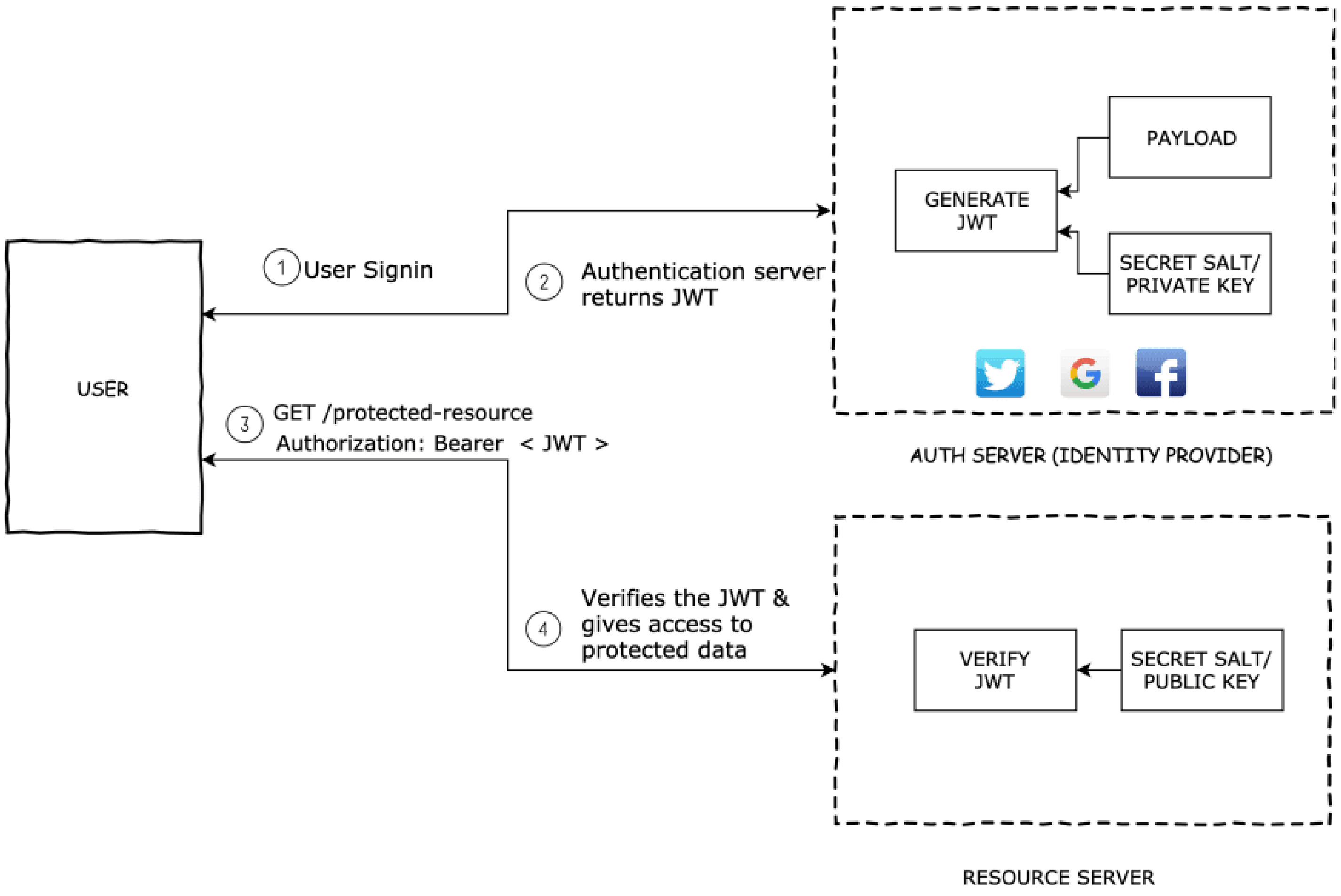
```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NT
Y3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5M
DIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o
```

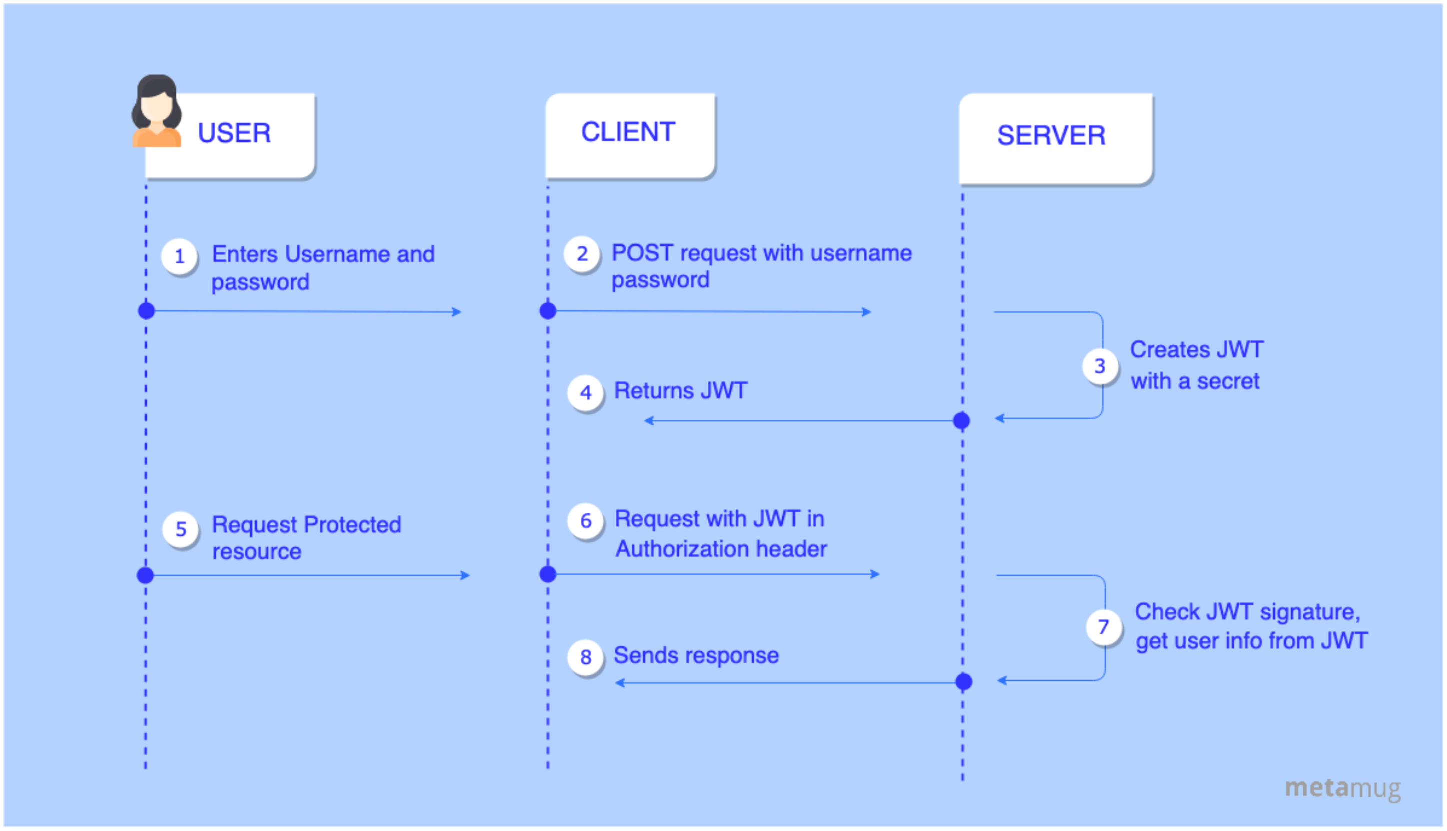
```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

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

```
HMACSHA256(
BASE64URL(header)
.
BASE64URL(payload),
secret)
```

Authentication Flow


USER
1 User Signin
2 Authentication server returns JWT
3 GET /protected-resource
Authorization: Bearer < JWT >
4 Verifies the JWT & gives access to protected data
GENERATE JWT
PAYLOAD
SECRET SALT/ PRIVATE KEY
AUTH SERVER (IDENTITY PROVIDER)
VERIFY JWT
SECRET SALT/ PUBLIC KEY
RESOURCE SERVER


USER
CLIENT
SERVER
1 Enters Username and password
2 POST request with username password
3 Creates JWT with a secret
4 Returns JWT
5 Request Protected resource
6 Request with JWT in Authorization header
7 Check JWT signature, get user info from JWT
8 Sends response
metamug

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():

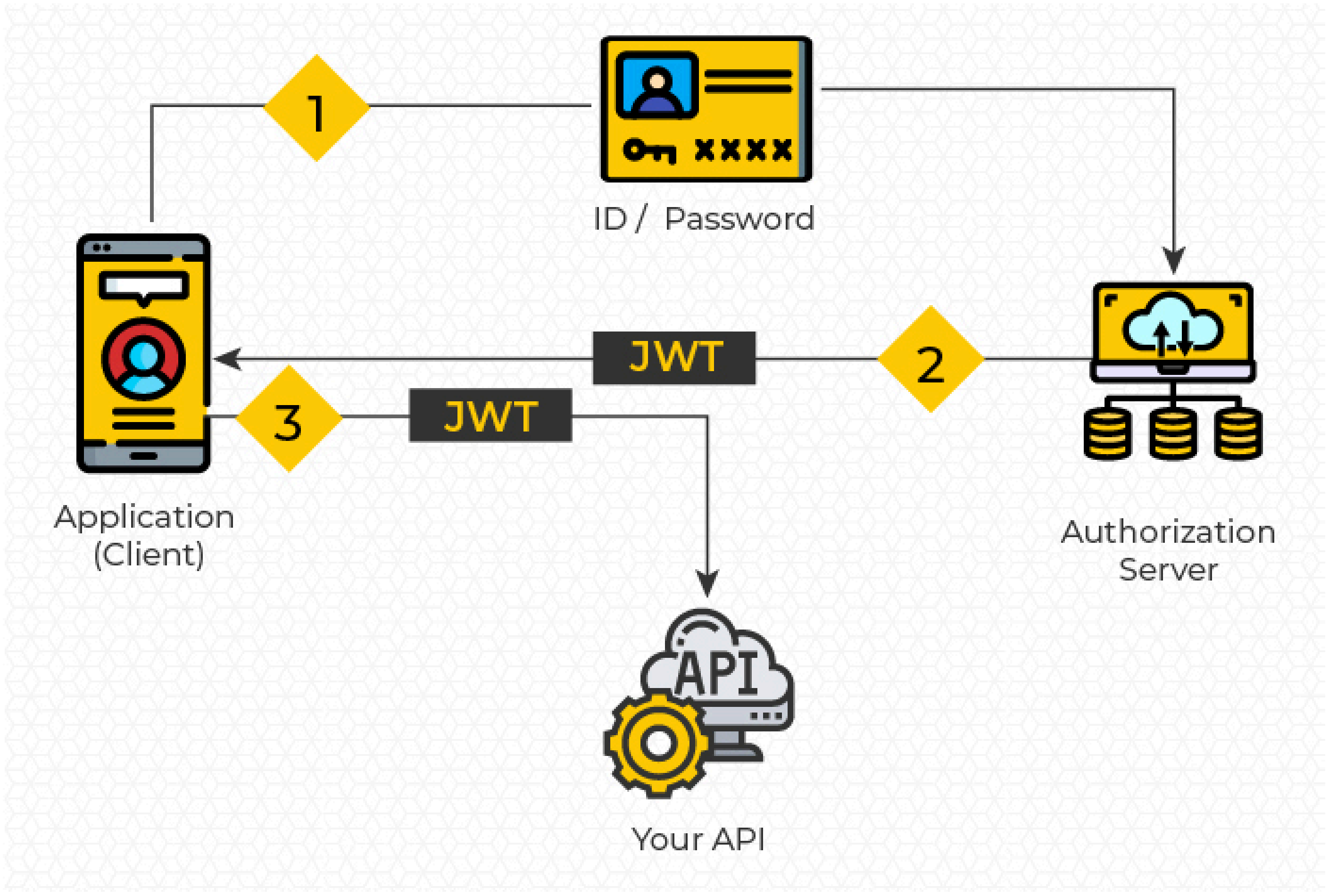
- This function is used to verify and decode a JWT token to retrieve the original payload.
- It takes three parameters:
    - `token`: The JWT token to be verified.
    - `secretOrPublicKey`: The secret key or public key used to verify the token's signature. If the token was signed using a symmetric 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.

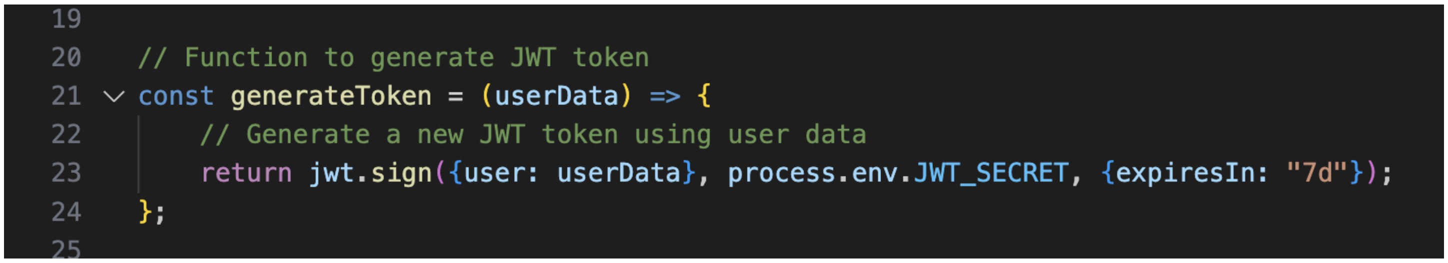
Once you've attached the decoded user information to `req.EncodedData` within the `jwtAuthMiddleware`, you can access it in further routes or middleware functions in your application

```
// 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

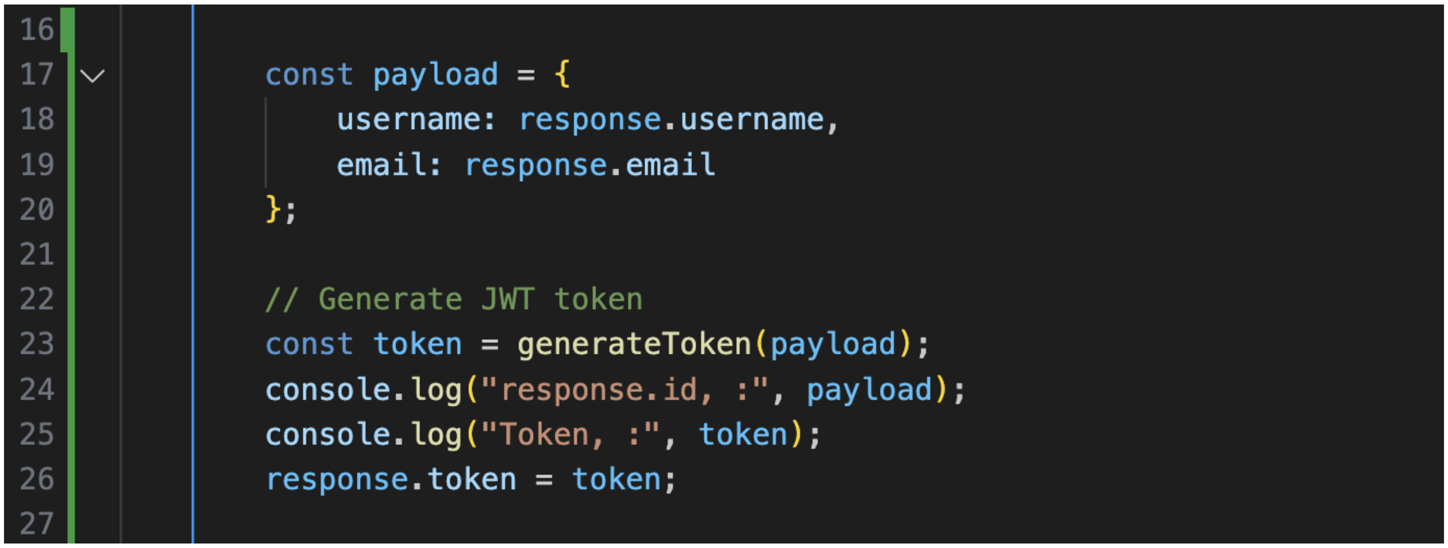
```
// Function to generate JWT token
const generateToken = (userData) => {
    // Generate a new JWT token using user data
    return jwt.sign({user: userData}, process.env.JWT_SECRET, { expiresIn: '1h'
});
};

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'`.



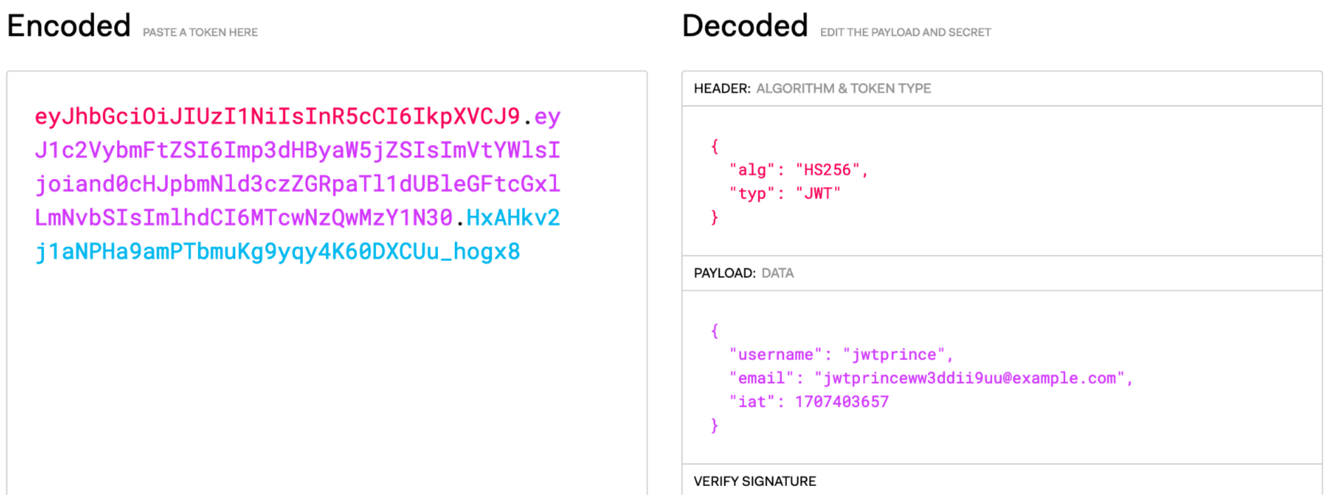
```javascript

// 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: "7d"});
};

```

- 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.



```javascript

    const payload = {
        username: response.username,
        email: response.email
    };

    // Generate JWT token
    const token = generateToken(payload);
    console.log("response.id, :", payload);
    console.log("Token, :", token);
    response.token = token;

```

Validate JWT in Encoder



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ey
J1c2VybmFtZSI6Imp3dHByaW5jZSIsImVtYWlsI
joiand0cHJpbmNld3czZGRpaTl1dUBleGFtcGxl
LmNvbSIsImlhdCI6MTcwNzQwMzY1N30.HxAHkv2
j1aNPHa9amPTbmuKg9yqy4K60DXCUu_hogx8
```

Decoded EDIT THE PAYLOAD AND SECRET

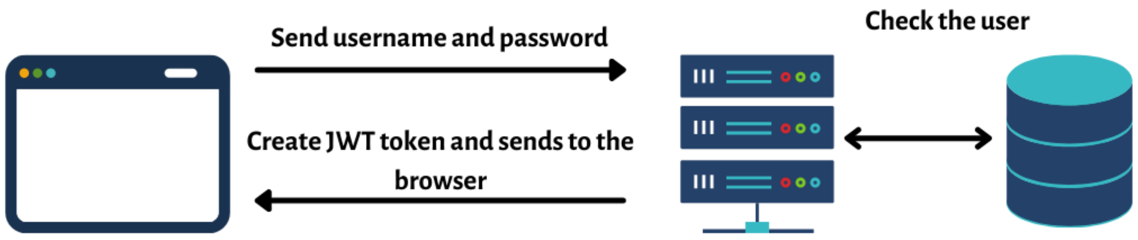
HEADER: ALGORITHM & TOKEN TYPE

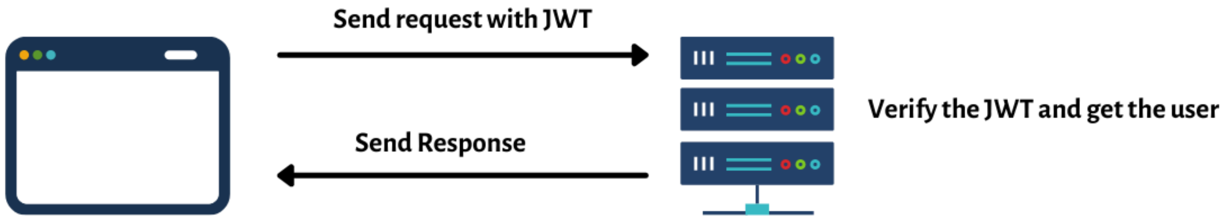
```json
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```json
{
  "username": "jwtprince",
  "email": "jwtprinceww3ddii9uu@example.com",
  "iat": 1707403657
}
```

VERIFY SIGNATURE


Send username and password
Check the user
Create JWT token and sends to the browser


Send request with JWT
Verify the JWT and get the user
Send Response

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.

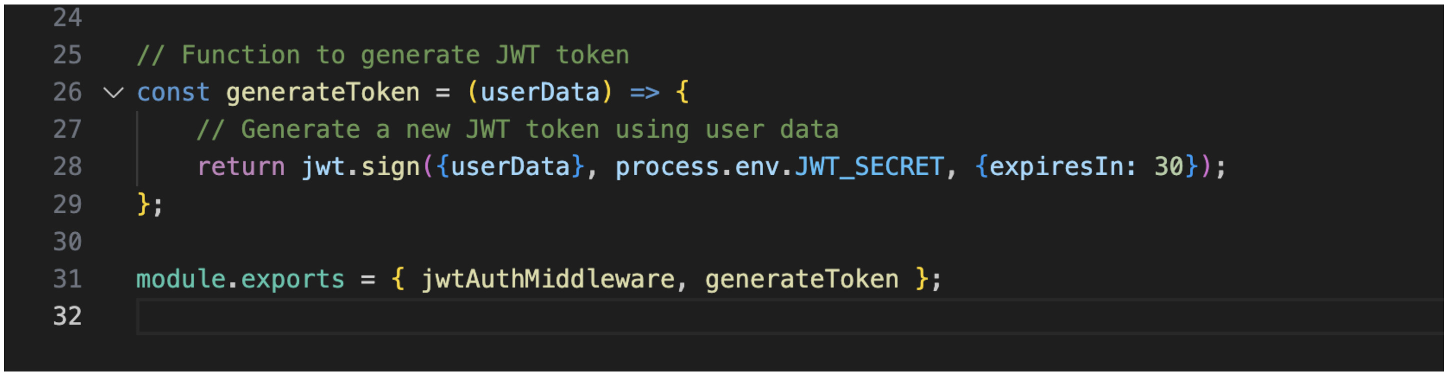
```
const authorizationHeader = 'Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmF
tZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJ
f36POk6yJV_adQssw5c';
const token = authorizationHeader.split(' ')[1];
console.log(token); // Output:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmF
tZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJ
f36POk6yJV_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.

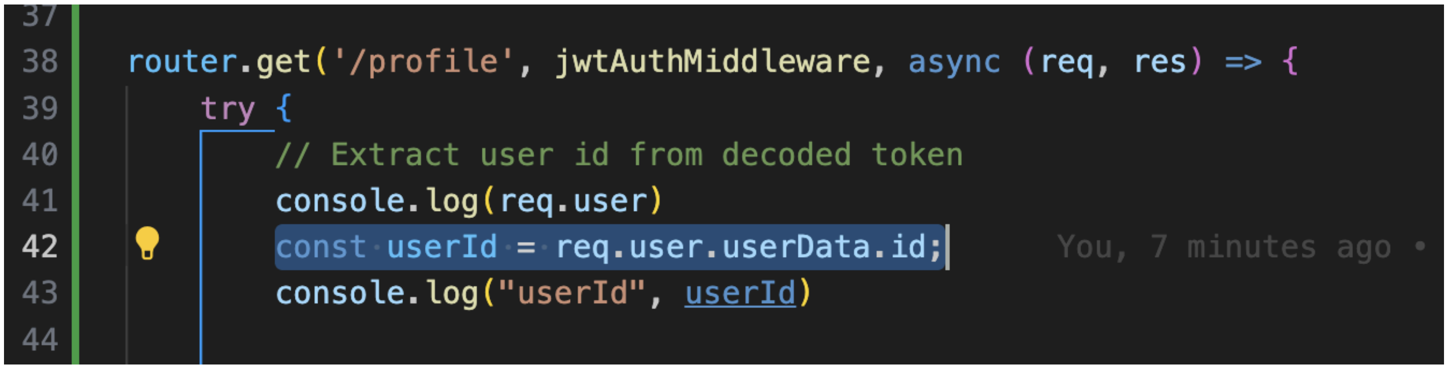
We can Add one more line of check in the jwtAuthMiddleware

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

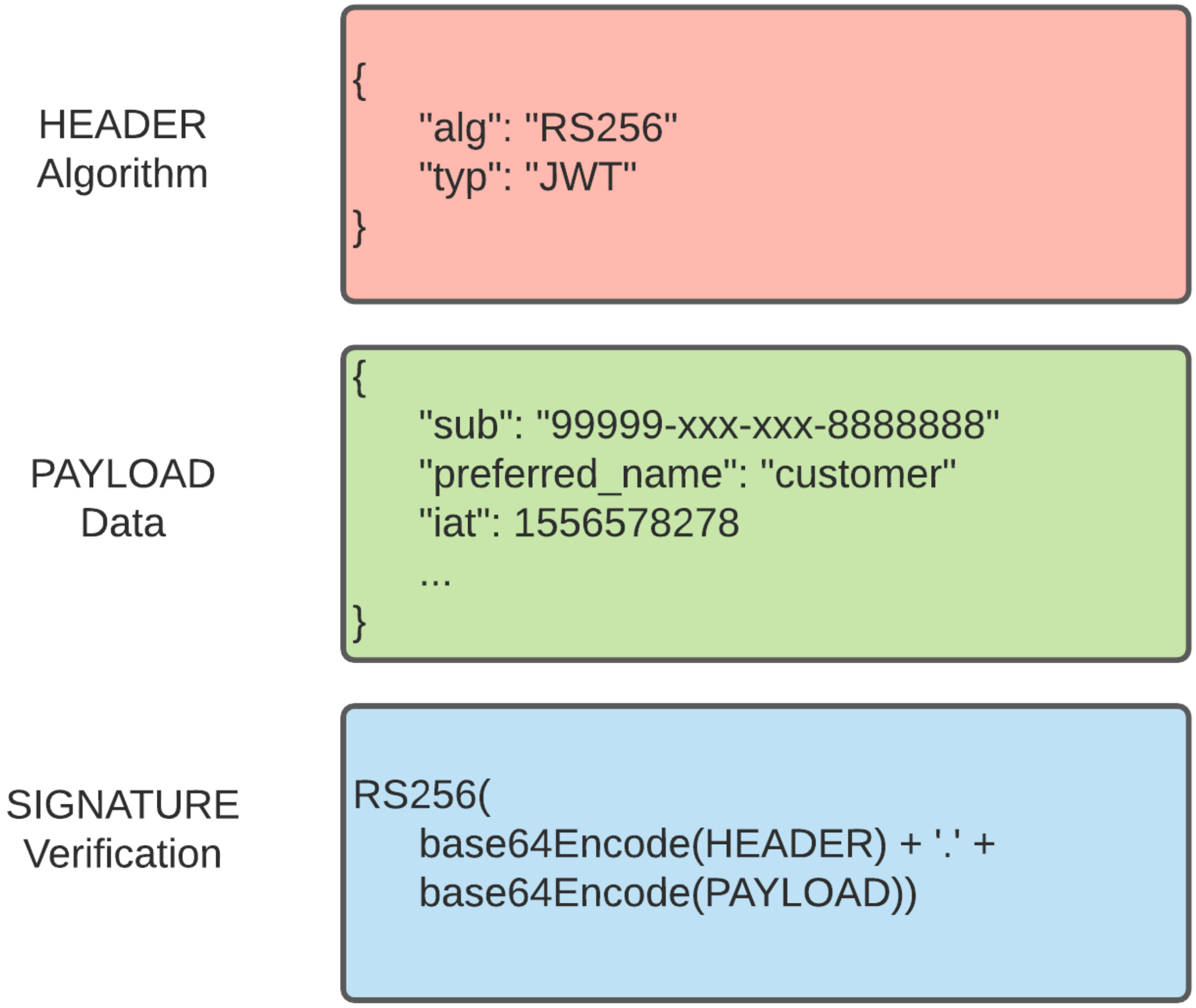
Expire Time-based Token



```

// Function to generate JWT token
const generateToken = (userData) => {
    // Generate a new JWT token using user data
    return jwt.sign({userData}, process.env.JWT_SECRET, {expiresIn: 30});
};

module.exports = { jwtAuthMiddleware, generateToken };

```



```
router.get('/profile', jwtAuthMiddleware, async (req, res) => {
    try {
        // Extract user id from decoded token
        console.log(req.user)
        const userId = req.user.userData.id;        You, 7 minutes ago •
        console.log("userId", userId)

```



HEADER
Algorithm
{
"alg": "RS256"
"typ": "JWT"
}
PAYLOAD
Data
{
"sub": "99999-xxx-xxx-8888888"
"preferred_name": "customer"
"iat": 1556578278
...
}
SIGNATURE
Verification
RS256(
base64Encode(HEADER) + '.' +
base64Encode(PAYLOAD))