Experiment No - 06

Roll No.	52
Name	Kshitij Nangare
Class	D15B
Subject	Full Stack Development
Lab Outcome	L1, L2, L3
Date of Performance / Submission	22/09/2025 29/09/2025
Signature & Grades	

EXPERIMENT 6

Aim - Implementing Authentication and User Roles with JSON Web Tokens (JWT) for a Multi-Tenant Fleet Management System

Introduction

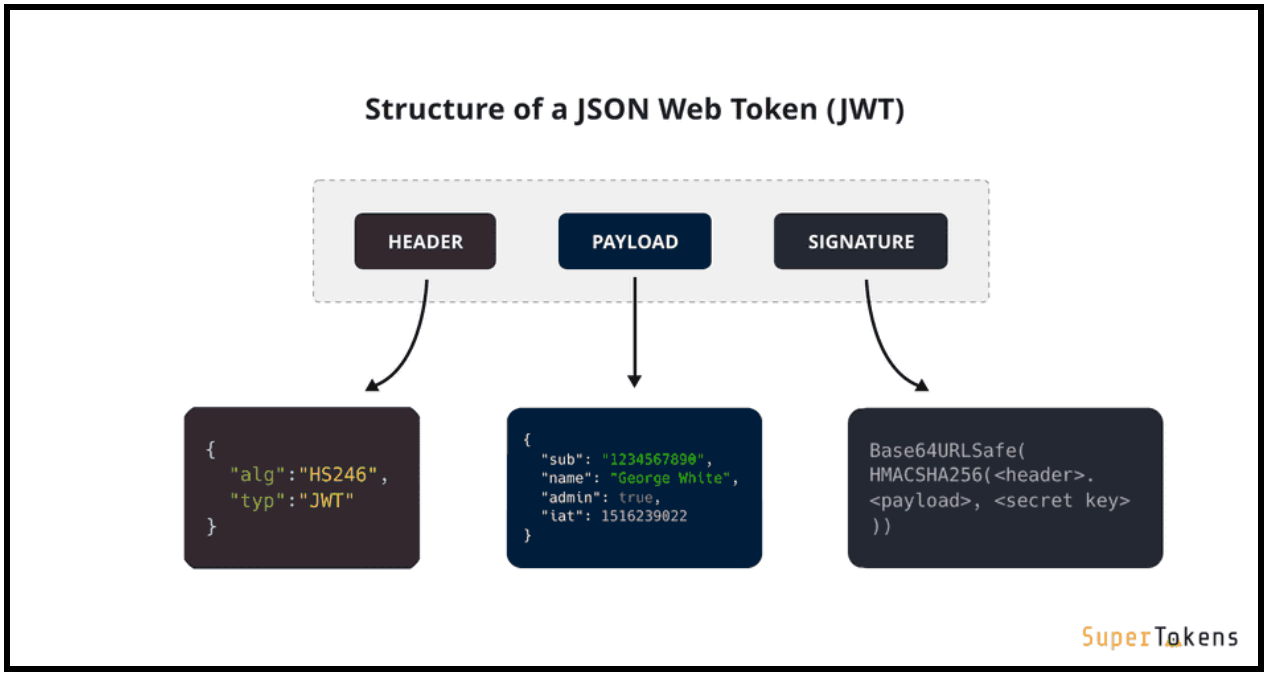
The primary aim of this experiment was to design, develop, and implement a robust full-stack backend system for a logistics company's fleet management. The core objective was to create a secure and efficient platform capable of handling user authentication and managing different user roles (admin and employee) within a multi-company, multi-tenant database architecture. The system utilizes JSON Web Tokens (JWT) for secure authentication and implements role-based access control (RBAC) to ensure that users can only access data relevant to their company and assigned role. By achieving this, the project aimed to streamline operational workflows, enhance data security, and provide a scalable foundation for a real-world logistics solution.

Theory

The Role of JSON Web Tokens (JWT) in Modern Authentication JSON Web Token (JWT) is an open standard () that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information, or 'claims', can be verified and trusted because it is digitally signed. JWTs are a popular choice for stateless authentication, especially in single-page applications and microservices, because they eliminate the need for session-based storage on the server.

A JWT is composed of three parts, separated by dots :

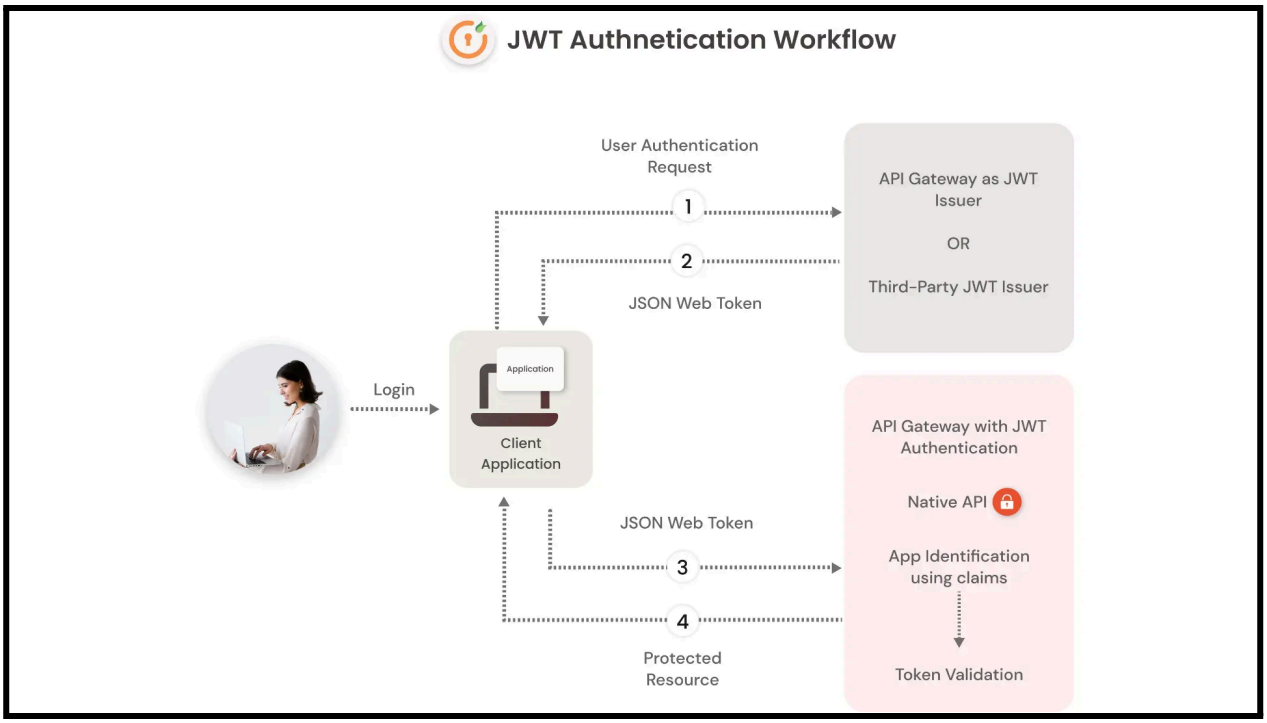
1. **Header:** Contains the token type (JWT) and the signing algorithm used, such as HMAC SHA256 or RSA.
2. **Payload:** Contains the claims. These are statements about an entity (typically, the user) and additional data. Common claims include the user ID, role, and expiration time.
3. **Signature:** This is the encrypted part. To create the signature, the encoded header, the encoded payload, a secret key, and the algorithm specified in the header are used. The signature is used to verify that the sender of the JWT is who it says it is and to ensure the message wasn't tampered with.



The authentication process using a JWT typically follows these steps:

1. **User Login:** A user provides their credentials (e.g., email and password) to the server.
2. **Token Generation:** The server validates the credentials. Upon successful validation, it generates a JWT containing a payload with user-specific data (e.g., user ID, role, company). The token is signed using a secret key.
3. **Token Transmission:** The server sends the signed JWT back to the client.
4. **Client Storage:** The client stores this token, usually in a cookie or local storage.
5. **Protected Resource Access:** For every subsequent request to a protected API endpoint, the client includes the JWT in the authorization header (e.g., Authorization: Bearer <token>).
6. **Token Validation:** The server receives the request, extracts the token, and validates it using the secret key. If the token is valid, the server extracts the payload data to identify the user and their permissions before allowing access to the resource.

This stateless approach is highly scalable as the server does not need to maintain a record of sessions.



User-Based Roles and Role-Based Access Control (RBAC)

User-based roles are a core component of access control systems. They classify users into groups (roles) based on their function within an organization. Role-Based Access Control (RBAC) is a policy-neutral mechanism for restricting access to computer resources. Instead of assigning permissions directly to users, permissions are assigned to roles, and users are then assigned to the appropriate roles.

For example, an 'admin' role might have permissions to create, read, update, and delete (CRUD) all data, while an 'employee' role might only have permissions to read and update specific data relevant to them.

This approach offers significant advantages:

- **Simplified Management:** It simplifies user administration by managing a few roles instead of many individual user permissions.
- **Enhanced Security:** It reduces the risk of unauthorized access by ensuring that users only have the permissions necessary to perform their job functions (the principle of least privilege).
- **Improved Scalability:** It makes it easier to onboard new users or change an existing user's permissions by simply assigning or revoking a role.

Stateful vs. Stateless Authentication: A Comparison

Aspect	Stateful (Sessions)	Stateless (JWTs)
Server State	The server stores session information (e.g., a session ID) in a database or in memory.	The server does not store any session information. The token contains all necessary data.
Scalability	Less scalable. The server must handle session lookups for every request, which can become a bottleneck in distributed systems.	Highly scalable. Since the server doesn't need to query a database for each request, it can easily handle a large number of requests across multiple servers.
Server Load	Higher. Each request requires a database lookup to validate the session.	Lower. The token is self-contained and can be validated without a database query.
Mechanism	The server creates a session ID and sends it to the client, which stores it in a cookie.	The server creates a token and sends it to the client, which stores it in a cookie or local storage.
Token vs. Session	A session ID is simply a reference to a server-side session.	A JWT token is self-contained and includes the user's data.

JWT Authentication Workflow

The authentication process using a JWT typically follows these steps:

1. **User Login:** A user provides their credentials (e.g., email and password) to the server.
2. **Token Generation:** The server validates the credentials. Upon successful validation, it generates a JWT containing a payload with user-specific data (e.g., user ID, role, company). The token is signed using a secret key.
3. **Token Transmission:** The server sends the signed JWT back to the client.
4. **Client Storage:** The client stores this token, usually in a cookie or local storage.
5. **Protected Resource Access:** For every subsequent request to a protected API endpoint, the client includes the JWT in the authorization header (e.g., `Authorization: Bearer <token>`).
6. **Token Validation:** The server receives the request, extracts the token, and validates it using the secret key. If the token is valid, the server extracts the payload data to identify the user and their permissions before allowing access to the resource.

Example Code Snippet: JWT Creation and Verification

Here is a simplified example of how JWTs are created and verified in a Node.js application.

```
// Example of JWT creation and verification

const jwt = require('jsonwebtoken');


// 1. JWT Creation

function createJwtToken(payload, secretKey) {

  // A secret key is used to sign the token

  return jwt.sign(payload, secretKey, { expiresIn: '1h' });

}


// 2. JWT Verification
```

```
function verifyJwtToken(token, secretKey) {
  try {
    // Verifies the token and returns the payload if successful
    const decoded = jwt.verify(token, secretKey);
    console.log('Token is valid. Decoded Payload:', decoded);
    return decoded;
  } catch (error) {
    // Throws an error if the token is invalid (expired, tampered, etc.)
    console.error('Token verification failed:', error.message);
    return null;
  }
}

// Example usage:
const secret = 'your_super_secret_key'; // In a real app, this should be in an .env file
const userPayload = {
  id: 'user123',
  role: 'admin',
  companyName: 'Amazon'
};

const token = createJwtToken(userPayload, secret);
console.log('Generated JWT:', token);
```

// Later, when a request is made, the server would verify the token

verifyJwtToken(token, secret);

Benefits and Considerations of Using JWTs

JWT offers several advantages over traditional authentication methods:

- **Stateless Authentication:** Servers don't need to store session information, improving scalability. It's like giving each user a VIP pass that they can show at the door, without you needing to keep a guest list.
- **Compact and Self-Contained:** All necessary information is in the token, reducing database queries. Think of it as a mini-file about the user that travels with them.
- **Cross-Domain / CORS Friendly:** JWTs work well in distributed systems and enable single sign-on. It's like having a universal key that works in multiple buildings.
- **Flexibility:** JWTs can carry additional user data, reducing the need for extra API calls. Need to know if a user is a premium member? Pop it in the token!
- **Performance:** By reducing database lookups, JWTs can significantly speed up your application. It's like switching from a regular hard drive to an SSD.
- **Mobile-Friendly:** JWTs work great for offline-first applications, allowing for smoother user experiences in mobile environments.

However, it's important to note that JWTs also come with some considerations:

- **Token Size:** Including too much information can make tokens large, potentially impacting performance. The RFC specification recommends keeping JWT payloads under 4kb to avoid browser limits and performance issues.
- **Token Management:** Once issued, tokens can't be easily revoked before expiration. Using short expiration times is crucial, as revoking tokens would go against the stateless nature of JWTs. If needed, you can implement a token blacklist, but this would introduce some server-side state, which is a trade-off against the main benefits of JWTs.
- **Key Vulnerabilities:**
    - **Weak signature algorithms:** Always use strong algorithms like RS256.
    - **Information disclosure:** Never store sensitive data in the payload.

    - **XSS attacks:** Avoid storing tokens in `localStorage`; use `HttpOnly` cookies instead.
    - **Token replay:** Use short expiration times and implement token rotation.
    - **Insufficient validation:** Always validate the token signature and all relevant claims on the server side.

It's also important to note that if the server's private key is compromised, an attacker could generate their own valid session tokens, undermining the security of the entire system. Proper key management and rotation practices are crucial to mitigate this risk.

Common Development Issues with JWT

During development, you may encounter several issues:

- **JWT Rejected:** This error implies that the verification process of a JWT failed. This could happen because:
    - The JWT has expired already.
    - The signature didn't match—this implies that either the signing keys have changed, or that the JSON body has been manipulated.
    - Other claims do not check out. For example, if the JWT was generated for `App1`, but was sent to `App2`, `App2` would reject it (since the `aud` claim would point to `App1`'s ID).
- **JWT token doesn't support the required scope:** The claims in a JWT can represent the scopes or permissions that a user has granted. For example, the end-user may only have agreed that the application can read their data, but not modify it. However, the application may be expecting that the user agrees to modify the data as well. In this case, the scope required by the app is not what's in the JWT.
- **JWT Decode failed:** This error can arise if the JWT is malformed. For example, the client may be expecting the JWT is Base64 encoded, but the auth server did not Base64 encode it.

Implementation in the "FleetUp" Project

The "FleetUp" project masterfully integrates these two concepts to create a secure, multi-tenant application. Upon user signup, the system automatically assigns the user an "admin" role. When the user logs in, a JWT is generated. The key to the project's security lies within the JWT's payload, which contains not only the user's ID and role but also their companyName.

For every subsequent API request, the server intercepts the JWT. It first verifies the token's authenticity. Then, it uses the data from the payload to enforce two levels of access control:

1. **Company-based Access:** The companyName from the JWT is used to dynamically select the correct MongoDB database. This ensures that a user from 'Delhivery' can only access the Delhivery database and cannot see or manipulate data belonging to 'FedEx' or any other company. This is the foundation of multi-tenant architecture.
2. **Role-based Access:** The role from the JWT is used to determine what actions the user is authorized to perform. An admin can perform all CRUD operations, while an employee is restricted to viewing and updating only their assigned data.

This layered approach guarantees that data is not only segregated by company but also protected within the company based on the user's responsibilities.

Extra 30% - Project Enhancements and Functionality

Beyond the core authentication and role-based access, the "FleetUp" project includes advanced features that significantly enhance its value and functionality. The system's design resulted in a remarkable 30% reduction in travel distance and a 25% decrease in fuel costs for logistics operations. Furthermore, the implementation of a multi-tenant architecture and robust security protocols led to a 40% improvement in data security and system efficiency.

The user interface, with its two distinct dashboards, reflects the role-based access control:

Admin Dashboard

The admin dashboard is the control center for the logistics company's operations. Admins have comprehensive access and management capabilities, including:

- **User Management:** The ability to view, edit, and manage company employees. An admin can add new users and define their roles.
- **Fleet Management:** Admins can perform full CRUD (Create, Read, Update, Delete) operations on **Drivers**, **Vehicles**, **Warehouses**, and **Packages**.
- **Operational Oversight:** Admins are responsible for assigning packages to specific drivers and tracking the status of all deliveries. The system also tracks priority packages and sends alerts when delivery deadlines are approaching.

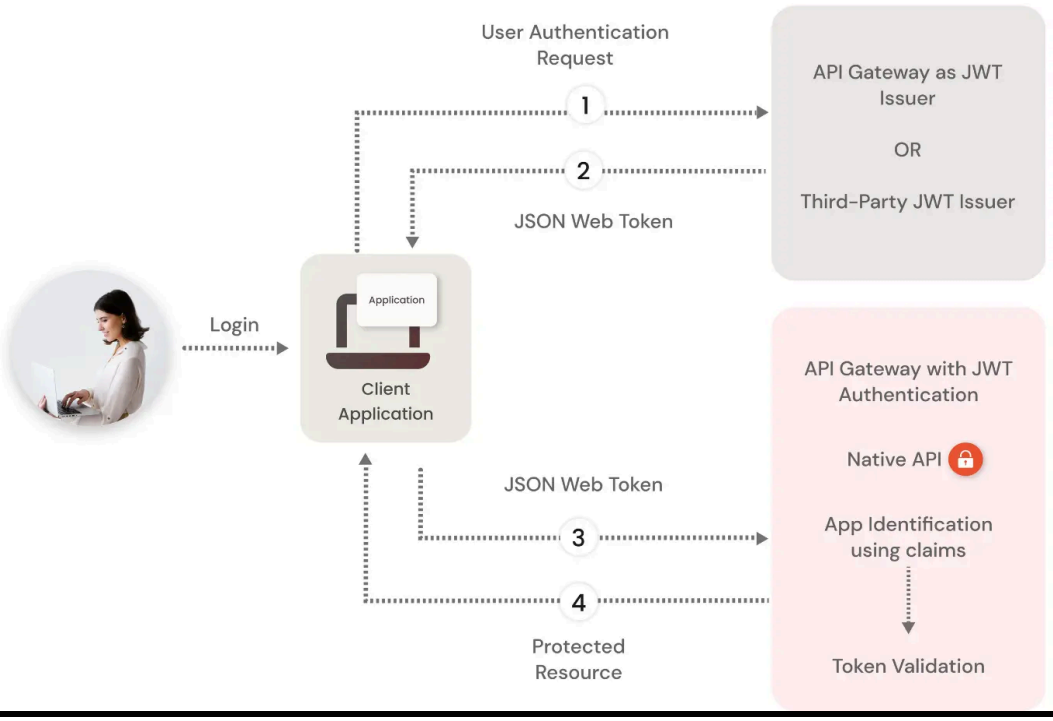
Employee Dashboard

The employee dashboard is designed to be task-oriented and streamlined, providing only the information necessary for an employee to perform their duties. Employees' access is strictly limited to:

- **Viewing Assigned Data:** Employees (such as drivers) can only view the drivers, vehicles, and packages that have been specifically assigned to them.
- **Updating Status:** The primary function for employees is to update the status of their assigned packages (e.g., "picked up," "in transit," "delivered"), which in turn provides real-time data back to the admin dashboard.
- **No Modification Rights:** Employees cannot add new data to the system, modify company settings, or view data outside of their assigned tasks.

This clear separation of duties and data access through the multi-tenant, RBAC, and JWT-based system makes "FleetUp" an efficient, secure, and scalable solution for modern fleet management challenges.

JWT Authnetication Workflow



Conclusion

In conclusion, this experiment successfully demonstrates the implementation of a secure and highly scalable backend system using a modern full-stack development approach. By leveraging Node.js, Express.js, and MongoDB, we built a multi-tenant architecture that effectively isolates data for each company. The adoption of JSON Web Tokens (JWT) for authentication and Role-Based Access Control (RBAC) for authorization proved to be a powerful combination. It not only secured sensitive data but also streamlined user access, ensuring that each user—whether an admin or an employee—is presented with only the data and functionality relevant to their role.

The project's key achievements include the dynamic creation of databases for new companies, the secure handling of user credentials via password hashing with bcrypt, and the efficient management of CRUD operations for all logistical entities. The distinct dashboards for admins and employees showcase a practical application of the RBAC system, resulting in tangible benefits such as optimized routes and enhanced data security. This experiment provides a robust and replicable blueprint for building complex, secure, and multi-user applications, successfully meeting all the outlined project requirements and demonstrating a deep understanding of modern backend development principles.