Introduction to Authentication and Authorization

Authentication and **Authorization** are essential for building secure web applications.

Authentication

- Process of verifying the identity of a user.
- Answers the question: **"Who are you?"**
- Example: Logging in with a username and password.

Authorization

- Process of determining what actions a user is allowed to perform.
- Answers the question: **"What can you do?"**
- Example: An admin can view all user data, while a regular user can only view their own data.

---

2. What is JWT (JSON Web Token)?

JWT is a compact, URL-safe token used for securely transmitting information between two parties: **client** and **server**.

Structure of a JWT

A JWT has three parts:

```
HEADER.PAYLOAD.SIGNATURE
```

Example:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiIxMjM0NTY3ODkwIiwicm9sZSI6ImFkbWluIiwiaWF0IjoxNjE4MjAwMDAwfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Header:

- Specifies metadata, such as the type of token (JWT) and signing algorithm (e.g., HS256).

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

Payload:

- Contains user-specific data (claims), such as userId, role, etc.

```
{
   "userId": "12345",
   "role": "admin",
   "iat": 1618200000
 }
```

Signature:

- Used to verify that the token was not tampered with.
- Created using

```
HMACSHA256(
   base64UrlEncode(header) + "." + base64UrlEncode(payload),
   secret
 )
```

How JWT Works

Workflow

1. **User Logs In**:
   - The user submits credentials (username/password) to the server.
2. **Server Verifies Credentials**:
   - If valid, the server generates a JWT and sends it to the client.
3. **Client Stores the Token**:
   - Token is stored in local storage or a cookie on the client.
4. **Client Sends Token**:
   - For each subsequent request, the client includes the token in the Authorization header.
5. **Server Validates Token**:
   - Server verifies the token's signature and decodes the payload to determine the user's identity and permissions.

4. Implementing JWT in Express.js

Installing Dependencies

Run the following command to install required packages:

```
npm install express jsonwebtoken bcryptjs body-parser
```

Registering a User

Create a route to register a new user. Use bcryptjs to hash the password before storing it

```
const express = require('express');
const bcrypt = require('bcryptjs');
const app = express();
app.use(express.json());

let users = []; // Simulated database

app.post('/register', async (req, res) => {
    const { username, password } = req.body;
    const hashedPassword = await bcrypt.hash(password, 10);
    users.push({ username, password: hashedPassword });
    res.status(201).json({ message: 'User registered successfully!' });
});
```

Logging in a User and Issuing a Token

When the user logs in, verify their credentials and generate a JWT.

```

const jwt = require('jsonwebtoken');
const SECRET_KEY = 'your_secret_key';

app.post('/login', async (req, res) => {
    const { username, password } = req.body;
    const user = users.find(u => u.username === username);

    if (!user || !(await bcrypt.compare(password, user.password))) {
        return res.status(401).json({ message: 'Invalid credentials' });
    }

    const token = jwt.sign({ username: user.username }, SECRET_KEY, { expiresIn: '1h' });
    res.json({ token });
});
```

Middleware to Secure Routes

Create middleware to verify the token

5. Securing Routes with JWT

Apply the authenticateToken middleware to protect routes:

```
app.get('/dashboard', authenticateToken, (req, res) => {
   res.json({ message: `Welcome, ${req.user.username}!` });
});
```

Role-Based Access Control

Adding Roles

Include roles in the JWT payload during login:

```
const token = jwt.sign({ username: user.username, role: user.role }, SECRET_KEY, {
expiresIn: '1h' });
```

Authorization Middleware

Create a middleware to restrict access based on roles:

```
const authorizeRole = (roles) => {
   return (req, res, next) => {
      if (!roles.includes(req.user.role)) {
         return res.status(403).json({ message: 'Access Denied' });
      }
      next();
   };
};
```

Protecting Role-Based Routes

```
app.get('/admin', authenticateToken, authorizeRole(['admin']), (req, res) => {
   res.json({ message: 'Welcome to the admin panel' });
});
```

Interview Questions and answers:

1. What is authentication, and how does it differ from authorization?

Answer:

- **Authentication** is the process of verifying the identity of a user.
  - Example: Logging in with a username and password.
- **Authorization** is the process of determining what actions a user is allowed to perform after authentication.

- Example: A user with an "admin" role can manage users, while a "guest" user can only view content.

2. What are common methods of authentication?

Answer:

1. **Password-based Authentication:** Users log in using a username and password.
2. **Token-based Authentication (e.g., JWT):** Users are issued a token upon successful login, which is used to authenticate subsequent requests.
3. **OAuth:** Third-party authentication (e.g., "Login with Google").
4. **Biometric Authentication:** Using fingerprints, facial recognition, etc.
5. **Multi-factor Authentication (MFA):** Combining two or more authentication factors (e.g., password + OTP).

---

3. Why is password hashing important?

Answer:

- Password hashing ensures that passwords are not stored in plain text.
- Even if the database is compromised, hashed passwords are difficult to reverse-engineer.
- Common hashing algorithms: **bcrypt**, **PBKDF2**, **Argon2**.

4 What are the key differences between session-based and token-based authentication?

Answer:

Feature	Session-based Authentication	Token-based Authentication (JWT)
Stateful/Stateless	Stateful (requires server memory)	Stateless (no server memory needed)
Storage	Stored in cookies	Stored in cookies or local storage
Scalability	Limited scalability	Highly scalable
Logout Handling	Easy (invalidate session)	Requires token invalidation logic

5. What are the common vulnerabilities in authentication systems?

Answer:

1. **Brute Force Attacks:** Guessing passwords repeatedly.
2. **SQL Injection:** Exploiting unvalidated inputs to retrieve sensitive data.
3. **Phishing:** Tricking users into sharing credentials.
4. **Session Hijacking:** Stealing session cookies.

5. **Cross-Site Scripting (XSS):** Injecting malicious scripts to steal tokens.

Questions on JWT

6. What is a JSON Web Token (JWT)?

Answer:

- A JWT is a compact, URL-safe token used to securely transmit information between parties as a JSON object.
- It is commonly used for **authentication** and **authorization** in stateless applications.

---

7. What are the main components of a JWT?

Answer:

1. **Header**: Contains metadata about the token (e.g., algorithm and type).
2. **Payload**: Contains claims (user-specific data, like id, role).
3. **Signature**: Verifies the token is not tampered with.

---

8. How does JWT authentication work?

Answer:

1. User logs in with credentials.
2. Server validates the credentials and issues a JWT.
3. Client stores the JWT (e.g., in local storage).
4. Client sends the token in the Authorization header for subsequent requests.
5. Server verifies the token before granting access.

---

9. How do you secure a JWT?

Answer:

1. Use strong secret keys for signing.
2. Set short token expiry times (expiresIn).
3. Use HTTPS to secure data transmission.
4. Store tokens securely (e.g., HttpOnly cookies).
5. Implement refresh tokens for prolonged sessions.

---

10. What is the difference between access tokens and refresh tokens?

Answer:

- **Access Token**:
  - Short-lived token used for authentication.
  - Expires quickly to reduce the risk of misuse.
- **Refresh Token**:
  - Long-lived token used to request a new access token without requiring re-login.
  - Stored securely and used sparingly.

---

11. What is the advantage of using JWT over traditional sessions?

Answer:

- **Stateless**: No need to store session data on the server.
- **Scalability**: Ideal for distributed systems and microservices.
- **Portable**: Can be used across multiple services without server-side coordination.

---

12. What are some common issues with JWT?

Answer:

1. **Token Size**: JWTs are larger than session IDs, which can increase bandwidth usage.
2. **No Built-in Revocation**: Once issued, a JWT is valid until it expires unless you implement a blacklist.
3. **Token Leakage**: If stored insecurely (e.g., in local storage), tokens are vulnerable to XSS attacks.

Implementation Questions

13. How would you implement authentication with JWT in Express.js?

Answer:

1. Install dependencies

```
npm install express jsonwebtoken bcryptjs
```

Sample Implementation

```
const express = require('express');
const jwt = require('jsonwebtoken');
const bcrypt = require('bcryptjs');
const app = express();
const SECRET_KEY = 'your_secret_key';
```

```
let users = [];

app.use(express.json());

// Register Route
app.post('/register', async (req, res) => {
   const { username, password } = req.body;
   const hashedPassword = await bcrypt.hash(password, 10);
   users.push({ username, password: hashedPassword });
   res.status(201).json({ message: 'User registered successfully!' });
});

// Login Route
app.post('/login', async (req, res) => {
   const { username, password } = req.body;
   const user = users.find(u => u.username === username);
   if (!user || !(await bcrypt.compare(password, user.password))) {
      return res.status(401).json({ message: 'Invalid credentials' });
   }
   const token = jwt.sign({ username: user.username }, SECRET_KEY, { expiresIn: '1h' });
   res.json({ token });
});

// Protected Route
const authenticateToken = (req, res, next) => {
   const token = req.headers['authorization'];
   if (!token) return res.status(403).json({ message: 'Token required' });
   jwt.verify(token, SECRET_KEY, (err, user) => {
      if (err) return res.status(401).json({ message: 'Invalid token' });
      req.user = user;
      next();
   });
};

app.get('/dashboard', authenticateToken, (req, res) => {
   res.json({ message: `Welcome, ${req.user.username}!` });
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

14. How do you secure routes in an Express.js application?

Answer:

1. Create a middleware to validate JWTs

```
const authenticateToken = (req, res, next) => {
   const token = req.headers['authorization'];
```

```
  if (!token) return res.status(403).json({ message: 'Token required' });
  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(401).json({ message: 'Invalid token' });
    req.user = user;
    next();
  });
};
```

Apply middleware to protected routes

```
app.get('/protected', authenticateToken, (req, res) => {
  res.json({ message: 'You have access to this route' });
});
```

15. How would you implement role-based access control in Express.js?

Answer:

1. Include roles in the JWT payload during login:

```
const token = jwt.sign({ username: user.username, role: user.role }, SECRET_KEY, {
expiresIn: '1h' });
```

Create an authorization middleware:

```
const authorizeRole = (roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({ message: 'Access Denied' });
    }
    next();
  };
};
```

Protect role-specific routes:

```
app.get('/admin', authenticateToken, authorizeRole(['admin']), (req, res) => {
  res.json({ message: 'Welcome to the admin panel' });
});
```

16. How would you handle token expiration in a real-world application?

Answer:

- Use short-lived access tokens and long-lived refresh tokens.
- Implement a /refresh endpoint to issue new tokens when the access token expires.

---

17. How would you invalidate a JWT?

Answer:

1. Use a token blacklist stored in a database or cache (e.g., Redis).
2. Check the token against the blacklist during each request.

---

18. What’s the difference between jsonwebtoken.sign() and jsonwebtoken.verify()?

Answer:

- sign(): Generates a token based on the payload and secret.
- verify(): Validates the token and decodes the payload.