📘 Detailed Notes: JWT, RBAC, useMemo and React.memo in Modern Web Development

🔐 JSON Web Tokens (JWT)

What is JWT?

JWT (JSON Web Token) is an open standard (RFC 7519) used for securely transmitting information between parties as a JSON object. It is compact, URL-safe, and widely used for authentication and authorization in modern web applications.

Structure of a JWT

A JWT consists of three parts:

1. Header:

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

2. Payload (Claims):

```json
{
  "sub": "1234567890",
  "name": "John Doe",
  "role": "admin",
  "iat": 1516239022,
  "exp": 1516242622
}
```

3. Signature: Created by encoding the header and payload using Base64Url encoding and signing it with a secret or private key.

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

Final Token:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwicm9sZSI6Im
```

Common Claims

- `sub` - Subject (user ID)
- `iat` - Issued at
- `exp` - Expiration time
- `role` - Custom claim for role-based access

Short-Lived vs Long-Lived JWTs

JWTs are generally categorized based on their lifespan:

Short-Lived Access Token

- Typically valid for a few minutes (e.g., 5 to 15 minutes)
- Reduces the window of exploitation if stolen
- Used for frequent API access

Example:

```
jwt.sign({ id: user.id }, secret, { expiresIn: '15m' });
```

Long-Lived Refresh Token

- Valid for days or weeks
- Stored securely (e.g., HttpOnly cookie or database)
- Used to obtain new access tokens without requiring the user to log in again

Example:

```
jwt.sign({ id: user.id }, refreshSecret, { expiresIn: '7d' });
```

Refresh Token Flow

1. User logs in and receives an access token and a refresh token.
2. Access token is used for authenticated API calls.
3. When the access token expires, client sends refresh token to a secure endpoint.
4. Server validates the refresh token and issues a new access token.

Server-side Refresh Endpoint Example

```
app.post('/refresh-token', (req, res) => {
  const refreshToken = req.body.token;
  if (!refreshToken) return res.sendStatus(401);
  jwt.verify(refreshToken, refreshSecret, (err, user) => {
    if (err) return res.sendStatus(403);
    const accessToken = jwt.sign({ id: user.id, role: user.role }, secret, { expiresIn: '15m' });
    res.json({ accessToken });
  });
});
```

Usage in Web Development

- User logs in → Server generates JWT → Client stores JWT (preferably in HttpOnly cookie)
- For each protected API request → JWT is sent in `Authorization: Bearer <token>` header
- Server verifies JWT → grants or denies access

Best Practices

- Use short-lived access tokens and refresh tokens
- Do not store JWTs in localStorage (XSS risk)
- Use strong secrets/keys

- Validate expiration (`exp`) and signature

💭 Role-Based Access Control (RBAC)

What is RBAC?

RBAC is a security approach to restrict system access to authorized users based on their roles.

Core Components

1. Users: Represent individuals in the system
2. Roles: Named collections of permissions (e.g., admin, editor, viewer)
3. Permissions: Allowed operations on resources (e.g., read, write, delete)

Principle of Least Privilege

Each user should have the minimum access necessary to perform their tasks.

JWT + RBAC Integration

- On login, user's role is embedded into the JWT payload.
- Middleware on server reads JWT → extracts role → validates access permissions for each route.

Example:

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

Backend Enforcement Example (Node.js + Express):

```
function checkRole(requiredRole) {
  return function(req, res, next) {
    const role = req.user.role;
    if (role !== requiredRole) return res.status(403).send('Access denied');
    next();
  }
}

app.get('/admin', authenticateJWT, checkRole('admin'), adminHandler);
```

Avoid These Mistakes

- Role enforcement on frontend only (easy to bypass)
- Using stale tokens when user roles change
- Not validating expiration and issuer

⚛️ useMemo and React.memo in React

What is React.memo?

React.memo is a higher-order component used to prevent unnecessary re-rendering of functional components when their props have not changed (shallow comparison).

Example:

```
const Greeting = React.memo(({ name }) => {
  console.log("Rendered");
  return <h1>Hello, {name}</h1>;
});
```

When to Use React.memo

- Functional components that render the same output for the same props
- Pure presentational components
- Large lists (with stable props)

What is useMemo?

useMemo is a React Hook used to memoize the result of an expensive computation to avoid recalculating it on every render unless its dependencies change.

Syntax:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

When to Use useMemo

- Expensive computations in render
- Filtering, sorting large datasets
- Avoiding unnecessary recalculation of derived state

Difference Between useMemo and React.memo

Feature	React.memo	useMemo
Type	HOC (wraps component)	Hook (inside component)
Memoizes	Render output	Computation result
Dependency compare	Shallow on props	Explicit via dependency array

Caveats

- Overuse can reduce performance due to overhead
- Always use useCallback when passing functions as props to memoized components

Example:

```
const handleClick = useCallback(() => setCount(c => c + 1), []);
```

✅ Summary Table

Concept	Purpose	Key Usage	Best Practice
JWT	Token-based stateless auth	API auth, session management	Use short expiry, refresh tokens
RBAC	Role-based access control	Protect routes based on user roles	Enforce on backend, use least privilege
React.memo	Memoize functional component output	Prevent re-renders with same props	Use for pure components
useMemo	Memoize computation result	Avoid recalculating expensive expressions	Use when needed; avoid excessive usage