Role-Based Access Control (RBAC) Implementation Report

Aim

To implement a secure and scalable **Role-Based Access Control (RBAC)** system within a web application, effectively managing and restricting user permissions across three distinct roles: **Admin**, **Moderator**, and **Basic User**.

Objective

The primary objectives are to:

1. **Define Role Hierarchy:** Establish and manage distinct permission levels for Admin (full control), Moderator (manage content/users), and User (read-only/self-modification).
2. **Integrate Role into JWT:** Include the user's assigned role directly within the **JSON Web Token (JWT)** payload upon login.
3. **Develop Authorization Middleware:** Create a reusable Express middleware function that checks the user's role (extracted from the JWT) against the required role for a specific route, granting or denying access.

Theory

Role-Based Access Control (RBAC)

RBAC is a security model where access permissions are dictated by the user's assigned role, not by the user's individual identity. This simplifies management: instead of assigning permissions to hundreds of users, you assign permissions to three or four roles, and then assign users to those roles.

The hierarchy for this implementation is:

- **Admin:** Has all permissions (Create, Read, Update, Delete) across all resources.
- **Moderator:** Can manage content and potentially basic users (e.g., Delete comments, Deactivate accounts) but cannot manage system settings or other Admins.
- **User:** Can only read general content and modify their own profile/content.

Implementation with JWT

To achieve fast and stateless authorization:

1. **Authentication:** Upon successful login, the server generates a JWT containing the user's id and their assigned **role** (e.g., role: 'admin').
2. **Authorization:** When a request hits a protected route, a two-part process occurs:
   - **Authentication Middleware (A):** The JWT is verified (signature checked) and the payload (including the role) is extracted and attached to the request object (req.user.role).

- **Authorization Middleware (B):** A second, specialized middleware checks req.user.role against a predefined list of roles allowed for that specific route.

Procedure (Using Node.js/Express)

1. **Enhance JWT Payload:** Modify the login function to ensure the user.role is included in the token payload before signing.
2. **Authentication Middleware:** Use the existing verifyToken middleware (from context) to attach req.user.role.
3. **Authorization Middleware:** Create a new function, checkRole(allowedRoles), which takes an array of permissible roles as an argument and executes the access check logic.
4. **Route Protection:** Apply both the authentication and the authorization middleware to the routes, passing the required roles to checkRole.

Code (Illustrative Node.js/Express Implementation)

This example assumes you have an existing user database and a verifyToken function that successfully validates a JWT and attaches the payload (e.g., req.user = { id: '...', role: 'user' }) to the request.

1. Authorization Middleware (authMiddleware.js)

```
// This function checks if the user's role (from the verified JWT) is
included
// in the array of allowed roles passed to the middleware.

/**
 * Middleware to check if the user has one of the allowed roles.
 * @param {string[]} allowedRoles - An array of roles that are permitted
to access the route.
 */
const checkRole = (allowedRoles) => {
    return (req, res, next) => {
        // Step 1: Ensure the user object (attached by the previous
verifyToken middleware) and role exist
        if (!req.user || !req.user.role) {
            // Should theoretically not happen if verifyToken runs first,
but ensures safety.
            return res.status(401).json({ message: "Access denied. Role
information missing." });
        }
        const userRole = req.user.role;

        // Step 2: Check if the user's role is in the list of allowed
roles
        if (allowedRoles.includes(userRole)) {
            // Role is allowed, proceed to the route handler
            next();
        } else {
            // Role is forbidden, return 403 Forbidden
            console.log(`User with role '${userRole}' attempted access to
restricted route.`);
            return res.status(403).json({
                message: "Forbidden. You do not have the necessary
permissions for this resource."
            });
        }
    };
};

module.exports = checkRole;
```

Output Description

The system's output is based on the HTTP status code returned upon accessing a protected endpoint:

User Role (via JWT)	Endpoint Accessed	Required Roles	Expected Status/Output
admin	/admin/user/101	['admin']	**200 OK** (Access Granted)
moderator	/admin/user/101	['admin']	**403 Forbidden** (Access Denied)

user	/posts/comment/500	['admin', 'moderator']	**403 Forbidden** (Access Denied)
moderator	/posts/comment/500	['admin', 'moderator']	**200 OK** (Access Granted)
user	/profile	['user', 'moderator', 'admin']	**200 OK** (Access Granted)
No Token	/admin/user/101	N/A	**401 Unauthorized** (Denied by verifyToken)

This clear status differentiation proves the authorization middleware is correctly inspecting the token's role and enforcing the access policy.

Learning Outcomes

1. **RBAC Implementation:** Students will learn the principles of **Role-Based Access Control (RBAC)** and successfully implement a system that maps permissions to roles rather than individual users, simplifying security management.
2. **Authorization Middleware:** The project demonstrates how to create dedicated **authorization middleware** that works in conjunction with authentication (JWT verification) to enforce granular access policies on a per-route basis.
3. **Stateful Security with Stateless Tokens:** Students will understand how to use a stateless technology like **JWT** to securely carry state information (the user's role) from the server to the client and back, enabling high-performance, verifiable authorization checks without database lookups on every request.