SECTION 10 — Role-Based Access Control (RBAC)

Basic RBAC vs Enterprise Permission Matrix

1. Authentication vs Authorization

Before understanding RBAC, it is important to distinguish between:

Authentication and Authorization.

- Authentication answers: Who is the user?
- Authorization answers: What is the user allowed to do?

In the previous section, we implemented authentication using JWT.

In this section, we implement authorization.

2. What Is RBAC?

RBAC stands for Role-Based Access Control.

It is a system where:

- Users are assigned roles.
- Roles determine what actions are allowed.

Example roles:

- ADMIN
- SELLER
- CUSTOMER

RBAC ensures that:

- Not all users can access all endpoints.
- Only authorized users can perform certain actions.

3. Basic RBAC (Beginner Approach)

The simplest RBAC implementation checks user role directly.

Example:

```
if (req.user.role !== "ADMIN") {
  throw new AppError("Access denied", 403);
}
```

Or in route:

```
router.post("/policy", protect, (req, res, next) => {
  if (req.user.role !== "ADMIN") {
    return next(new AppError("Access denied", 403));
  }
  next();
});
```

Problems With Basic RBAC

1. Hardcoded role checks everywhere.
2. Not scalable when roles increase.
3. Difficult to manage large permission sets.
4. No flexibility for multiple roles.
5. Business logic becomes messy.

In small applications, this may work.
In enterprise systems, this approach fails.

4. Enterprise RBAC — Permission Matrix Approach

Instead of directly checking roles, enterprise systems use:

Permissions

Roles are mapped to permissions.

Routes require permissions, not roles.

This creates a flexible system.

5. permission.matrix.js

This file defines:

1. All available permissions.
2. Mapping of roles to permissions.

Step 1 — Define Permissions

```
export const PERMISSIONS = {
  MANAGE_PLATFORM_POLICY: "MANAGE_PLATFORM_POLICY",
  VIEW_PLATFORM_POLICY: "VIEW_PLATFORM_POLICY"
};
```

Each permission represents an action in the system.

Step 2 — Map Roles to Permissions

```
export const ROLE_PERMISSIONS = {
  ADMIN: [
    PERMISSIONS.MANAGE_PLATFORM_POLICY,
    PERMISSIONS.VIEW_PLATFORM_POLICY
  ],
  SELLER: [],
  CUSTOMER: []
};
```

Now:

- ADMIN can manage and view policy.
- SELLER and CUSTOMER cannot.

6. Permission Middleware (permission.middleware.js)

This middleware checks whether the user has required permission.

Example Implementation

```
import { ROLE_PERMISSIONS } from "../config/permission.matrix.js";
import AppError from "../utils/appError.util.js";

const authorize = (...requiredPermissions) => {
  return (req, res, next) => {
    const userRoles = req.user.roles;

    const userPermissions = userRoles.flatMap(
      role => ROLE_PERMISSIONS[role] || []
    );

    const hasPermission = requiredPermissions.every(permission =>
      userPermissions.includes(permission)
    );

    if (!hasPermission) {
      return next(new AppError("Access denied", 403));
    }

    next();
  };
};

export default authorize;
```

7. How Authorization Flow Works

When a protected route is accessed:

```
Request
  ↓
protect middleware (authentication)
  ↓
authorize middleware (permission check)
  ↓
Controller
```

If user lacks permission:

- 403 Forbidden response is returned.

8. Route Example Using Permission Matrix

```
router.post(
  "/policy",
  protect,
  authorize(PERMISSIONS.MANAGE_PLATFORM_POLICY),
  createPolicy
);
```

This means:

- User must be authenticated.
- User must have MANAGE_PLATFORM_POLICY permission.

We do not check for role directly.

9. Why Permission Matrix Is Better

Permission matrix provides:

1. Flexibility
2. Scalability
3. Centralized control
4. Easier updates
5. Support for multiple roles per user

Example:

If a new role "SUPER_ADMIN" is introduced:

We only update permission.matrix.js.

No route changes required.

10. Support for Multiple Roles

In our system:

roles: ["ADMIN"]

But in enterprise systems, a user may have:

roles: ["ADMIN", "SELLER"]

The permission middleware merges permissions of all roles.

This makes the system powerful and flexible.

11. Authorization Failure Response

If permission check fails:

```
{
  "success": false,
  "message": "Access denied",
  "errorCode": "FORBIDDEN"
}
```

Standardized via centralized error handling.

12. Enterprise-Level Authorization Benefits

Permission matrix approach:

- Avoids role hardcoding.
- Supports future growth.
- Supports feature-based permission design.
- Cleanly separates identity from authorization.
- Enables policy-driven systems.

This approach is used in large-scale backend systems.

13. Comparison: Basic RBAC vs Permission Matrix

Feature	Basic RBAC	Permission Matrix
Role Check	Direct	Indirect via permissions
Scalability	Low	High
Flexibility	Limited	Flexible
Centralized Control	No	Yes
Multi-Role Support	Weak	Strong

14. What Students Must Understand

1. Authentication verifies identity.
2. Authorization verifies permissions.
3. Roles are not permissions.
4. Permission matrix enables scalable access control.
5. Middleware enforces security before controller logic.
6. Enterprise systems avoid hardcoded role checks.

15. Summary

By the end of this section, students should understand:

- What RBAC is
- How basic RBAC works
- Why basic RBAC is limited
- How enterprise permission matrix works
- How authorization middleware enforces rules
- Why this approach is scalable

This completes the Authorization layer foundation.