# Hotel Booking System - Authorization Documentation

**Author:** Aswinnath TE | **Date:** November 9, 2025

## 📋 Table of Contents

1. Role-Based Access Control (RBAC)
2. Permission Model
3. User Dependencies & Decorators
4. Authorization Best Practices
5. Troubleshooting
6. Implementation Checklist
7. Reference: Common Permission Checks

**Note:** This document focuses on **Authorization** only. For authentication topics (JWT, tokens, login), see `AUTHENTICATION.md` .

# Role-Based Access Control (RBAC)

## Role Hierarchy

```
┌─────────────────────────────────────────┐
│              System Roles                │
├─────────────────────────────────────────┤
│                                          │
│  1. customer                             │
│     ├─ role_id: 1                        │
│     ├─ Permissions: BOOKING:READ/WRITE  │
│     └─ Restrictions: Can only access    │
│        their own bookings/reviews        │
│                                          │
│  2. normal_admin                         │
│     ├─ role_id: 3                        │
│     ├─ Permissions: Multiple            │
│     └─ Restrictions: Limited admin ops  │
│                                          │
│  3. super_admin                          │
│     ├─ role_id: 2                        │
│     ├─ Permissions: All resources        │
│     └─ Restrictions: None                │
│                                          │
└─────────────────────────────────────────┘
```

# Role Creation & Management ( `app/routes/roles_and_permissions_management/` )

## Create Role

```
@roles_and_permissions_router.post("/", response_model=RoleResponse)
async def create_new_role(
    payload: RoleCreate,
    db: AsyncSession = Depends(get_db),
    _ok: bool = Depends(ensure_not_basic_user),  # Must not be customer
    user_perms: dict = Depends(get_user_permissions),
):
    """
    Create a new role

    Requires: ADMIN_CREATION:READ permission
    """
    if "ADMIN_CREATION" not in user_perms or "READ" not in user_perms["ADMIN_CREATION"]:
        raise HTTPException(status_code=403, detail="Insufficient privileges")

    # Create role and audit log
```

## Assign Permissions to Role

```
@roles_and_permissions_router.post("/assign", response_model=RolePermissionResponse)
async def assign_permissions_to_role(
    payload: RolePermissionAssign,  # {role_id, permission_ids}
    db: AsyncSession = Depends(get_db),
    _ok: bool = Depends(ensure_not_basic_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """
    Assign multiple permissions to a role

    Requires: ADMIN_CREATION:READ permission
    """
    # Validate permission
    # Create mappings in permission_role_map table
    # Invalidate cache
    # Log audit trail
```

## Permission Query

```python
@roles_and_permissions_router.get("/permissions")
async def get_permissions(
    permission_id: int | None = Query(None),  # Get roles for this permission
    role_id: int | None = Query(None),        # Get permissions for this role
    resources: List[str] | None = Query(None), # Get permissions for resources
    db: AsyncSession = Depends(get_db),
    _ok: bool = Depends(ensure_not_basic_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """
    Query permissions in flexible ways:

    Example 1: Get all permissions for a role
      GET /roles/permissions?role_id=2

    Example 2: Get all roles that have a permission
      GET /roles/permissions?permission_id=5

    Example 3: Get permissions for specific resources
      GET /roles/permissions?resources=BOOKING&resources=ROOM_MANAGEMENT
    """
    # Validate that exactly one filter is provided
    # Query database
    # Return results
```

# Permission Model

## Resource Types

Permission resources represent features/domains in the system:

```
class Resources(str, PyEnum):
    BOOKING = "BOOKING"                    # Booking operations
    ADMIN_CREATION = "ADMIN_CREATION"      # User & role management
    ROOM_MANAGEMENT = "ROOM_MANAGEMENT"    # Room CRUD & inventory
    PAYMENT_PROCESSING = "PAYMENT_PROCESSING"  # Payment transactions
    REFUND_APPROVAL = "REFUND_APPROVAL"    # Refund operations
    CONTENT_MANAGEMENT = "CONTENT_MANAGEMENT"  # CMS content
    ISSUE_RESOLUTION = "ISSUE_RESOLUTION"  # Support tickets
    NOTIFICATION_HANDLING = "NOTIFICATION_HANDLING"  # Notifications
    ANALYTICS_VIEW = "ANALYTICS_VIEW"      # Reports & analytics
    BACKUP_OPERATIONS = "BACKUP_OPERATIONS"  # Database backups
    RESTORE_OPERATIONS = "RESTORE_OPERATIONS"  # Database restores
    OFFER_MANAGEMENT = "OFFER_MANAGEMENT"  # Promotions
```

## Permission Types

Each resource can have multiple permission levels:

```
class PermissionTypes(str, PyEnum):
    READ = "READ"      # View/fetch data
    WRITE = "WRITE"    # Create/update data
    DELETE = "DELETE"  # Remove data
    MANAGE = "MANAGE"  # Administrative control
    APPROVE = "APPROVE"  # Approval workflows
    EXECUTE = "EXECUTE"  # Execute operations (backups, etc.)
```

## Default Permission Assignments ( `database/seed_data.py` )

```
PERMISSIONS = [
    # BOOKING
    (5, 'BOOKING', 'READ'),
    (6, 'BOOKING', 'WRITE'),
    (7, 'BOOKING', 'DELETE'),
    (8, 'BOOKING', 'MANAGE'),
    (9, 'BOOKING', 'APPROVE'),
    (10, 'BOOKING', 'EXECUTE'),

    # ADMIN_CREATION
    (11, 'ADMIN_CREATION', 'READ'),
    (12, 'ADMIN_CREATION', 'WRITE'),
    (13, 'ADMIN_CREATION', 'DELETE'),
    (14, 'ADMIN_CREATION', 'MANAGE'),
    (15, 'ADMIN_CREATION', 'APPROVE'),
    (16, 'ADMIN_CREATION', 'EXECUTE'),

    # ROOM_MANAGEMENT
    (17, 'ROOM_MANAGEMENT', 'READ'),
    (18, 'ROOM_MANAGEMENT', 'WRITE'),
    # ... more permissions
]

# Default role-permission mappings
ROLE_PERMISSIONS = {
    1: [5, 6],  # customer: BOOKING:READ, BOOKING:WRITE
    2: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, ...],  # super_admin: all
    3: [5, 6, 7, 8, 11, 12, 13],  # normal_admin: limited
}
```

# Permission Lookup

```
async def get_user_permissions(
    current_user: Users = Depends(get_current_user),
    db: AsyncSession = Depends(get_db),
) -> dict[str, set[str]]:
    """
    Build permission dictionary for current user

    Returns:
    {
        "BOOKING": {"READ", "WRITE"},
        "ADMIN_CREATION": {"READ"},
        "ROOM_MANAGEMENT": {"READ"},
        ...
    }

    Query Flow:
    1. Get user's role_id from JWT token
    2. JOIN roles → permission_role_map → permissions
    3. Build {resource: set(permission_types)}
    4. Cache result (TTL: 5 min)
    5. Return to route for access control
    """
    role_id = current_user.role_id

    result = await db.execute(
        select(Permissions.resource, Permissions.permission_type)
        .join(PermissionRoleMap, PermissionRoleMap.permission_id == Permissions.permission_id)
        .where(PermissionRoleMap.role_id == role_id)
    )

    records = result.all()

    permissions_map = {}
    for resource, perm_type in records:
        resource_key = str(resource.value if hasattr(resource, 'value') else resource).upper()
        perm_key = str(perm_type.value if hasattr(perm_type, 'value') else perm_type).upper()
        permissions_map.setdefault(resource_key, set()).add(perm_key)

    return permissions_map
```

# User Dependencies & Decorators

## Core Dependencies

### 1. `get_current_user()`

Extracts authenticated user from JWT token

```
@app.get("/profile/me")
async def get_profile(current_user: Users = Depends(get_current_user)):
    """Returns currently authenticated user's profile"""
    return UserResponse.model_validate(current_user)
```

### 2. `get_user_permissions()`

Gets permission dictionary for current user

```
@app.get("/bookings/")
async def get_bookings(
    current_user: Users = Depends(get_current_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """Permission-aware booking query"""
    if "BOOKING" not in user_perms:
        raise ForbiddenError("No booking access")
```

### 3. `ensure_not_basic_user()`

Rejects requests from basic customer role (role_id == 1)

```
async def ensure_not_basic_user(
    current_user: Users = Depends(get_current_user)
) -> bool:
    """Dependency that rejects basic user role (role_id == 1)"""
    if getattr(current_user, "role_id", None) == 1:
        raise HTTPException(
            status_code=403,
            detail="Insufficient privileges: action not available for basic users"
        )
    return True


# Usage in route
@app.post("/roles/")
async def create_role(
    payload: RoleCreate,
    db: AsyncSession = Depends(get_db),
    _ok: bool = Depends(ensure_not_basic_user),  # Enforced here
):
    """Only admins can create roles"""
```

## 4. `ensure_only_basic_user()`

Permits only basic customer role

```
async def ensure_only_basic_user(
    current_user: Users = Depends(get_current_user)
) -> bool:
    """Dependency that permits only customer role (role_id == 1)"""
    if getattr(current_user, "role_id", None) != 1:
        raise HTTPException(
            status_code=403,
            detail="This action is only available for basic users"
        )
    return True


# Usage in route
@app.post("/bookings/")
async def create_booking(
    payload: BookingCreate,
    _basic_user: bool = Depends(ensure_only_basic_user),  # Only customers
):
    """Customers can book rooms"""
```

### 5. `ensure_admin()`

Verifies admin role by name (from roles_utility table)

```
async def ensure_admin(
    current_user: Users = Depends(get_current_user),
    db: AsyncSession = Depends(get_db)
) -> bool:
    """Dependency that permits only 'ADMIN' role"""
    role_id = getattr(current_user, "role_id", None)
    if not role_id:
        raise HTTPException(status_code=403, detail="Insufficient privileges")

    role = await db.execute(
        select(Roles).where(Roles.role_id == role_id)
    )
    role_obj = role.scalars().first()

    if not role_obj or role_obj.role_name.upper() != "ADMIN":
        raise HTTPException(status_code=403, detail="Admin privileges required")

    return True
```

# Permission-Based Access Control

## Pattern 1: Check Specific Permission

```
@app.post("/bookings/")
async def create_booking(
    payload: BookingCreate,
    current_user: Users = Depends(get_current_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """Require BOOKING:WRITE permission"""
    if not (
        Resources.BOOKING.value in user_perms
        and PermissionTypes.WRITE.value in user_perms[Resources.BOOKING.value]
    ):
        raise ForbiddenError("Insufficient permissions to create bookings")
```

## Pattern 2: Check ANY Permission from Resource

```
@app.get("/bookings/")
async def get_bookings(
    current_user: Users = Depends(get_current_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """Require any booking permission"""
    if Resources.BOOKING.value not in user_perms:
        raise ForbiddenError("No booking access")
```

### Pattern 3: Multiple Permission Check

```
@app.put("/refunds/{id}/approve")
async def approve_refund(
    refund_id: int,
    current_user: Users = Depends(get_current_user),
    user_perms: dict = Depends(get_user_permissions),
):
    """Require REFUND_APPROVAL:APPROVE permission"""
    if not (
        Resources.REFUND_APPROVAL.value in user_perms
        and PermissionTypes.APPROVE.value in user_perms[Resources.REFUND_APPROVAL.value]
    ):
        raise ForbiddenError("Insufficient privileges to approve refunds")
```

# Authorization Best Practices

## 1. Access Control

```
✅ DO:
  • Use role-based access control (RBAC)
  • Check permissions on every protected route
  • Use fine-grained permissions (resource + action)
  • Log all permission denials
  • Audit sensitive operations
  • Validate user ownership (own bookings, etc.)

❌ DON'T:
  • Rely only on client-side checks
  • Skip permission checks for "public" operations
  • Trust client-provided user_id
  • Use permission bypass for debugging
  • Mix authentication and authorization
```

# 4. API Security

```
✅ DO:
  • Validate all inputs (Pydantic)
  • Use parameterized queries (SQLAlchemy ORM)
  • Rate limit endpoints
  • Enable CORS with specific origins
  • Add CSRF protection if needed
  • Sanitize error messages
  • Log security events
  • Implement audit trails

❌ DON'T:
  • Accept arbitrary input without validation
  • Use string interpolation in queries
  • Return sensitive data in errors
  • Allow unlimited request rates
  • Trust Content-Type headers
  • Disable CORS security
```

# 2. Audit Logging

```
Every sensitive operation should be logged:
- User login/logout
- Permission changes
- User creation/deletion
- Booking creation/cancellation
- Payment processing
- Refund approval
- File uploads
- Configuration changes

Logs include:
- WHO: user_id, email
- WHAT: resource, action, old_value, new_value
- WHEN: timestamp
- WHERE: endpoint, IP address
- WHY: reason/context
```

# Troubleshooting

## 1. Permission Check Always Failing

Problem: Route returns 403 even with correct role

Causes:

- Permission name case mismatch
- Permission not assigned to role
- get_user_permissions() returns empty dict
- Resource enum name wrong

Solution:

- Use uppercase names: Resources.BOOKING.value
- Check permission_role_map table
- Debug: print user_perms in route
- Verify seed_data.py ran successfully

# Implementation Checklist

When implementing a new protected endpoint:

- □ Define required permission (Resource + PermissionType)
- □ Add dependency: `user_perms: dict = Depends(get_user_permissions)`
- □ Add permission check in route handler
- □ Test with user lacking permission (should get 403)
- □ Test with user having permission (should succeed)
- □ Add to API documentation
- □ Log sensitive operations with audit helper
- □ Cache permission-related queries
- □ Handle permission changes (user re-login to refresh)

# Reference: Common Permission Checks

## Check 1: Single Permission Required

```
if not (
    Resources.BOOKING.value in user_perms
    and PermissionTypes.WRITE.value in user_perms[Resources.BOOKING.value]
):
    raise ForbiddenError("Insufficient permissions")
```

## Check 2: Any Permission in Resource

```
if Resources.BOOKING.value not in user_perms:
    raise ForbiddenError("No booking access")
```

## Check 3: Role-Based (Not Recommended)

```
role_id = getattr(current_user, "role_id", None)
if role_id not in [2, 3]:  # super_admin, normal_admin
    raise ForbiddenError("Admin only")
```

## Check 4: Ownership Check

```
if booking.user_id != current_user.user_id:
    raise ForbiddenError("Cannot access other user's booking")
```

**Last Updated**: November 2025

**Version**: 1.0.0