

Authorization

Why Authorization is Important

- Authorization = the process of determining what a user can and cannot do within a system.
- Authentication vs. Authorization: Who you are vs. what you can do.
- Authorization is crucial for security and proper functioning of any application.
- Risks: unauthorized data access, data manipulation, privilege escalation.
- Business impact: financial loss, reputational damage, legal implications.

Part 1: Understanding RBAC

- **Overview:** Define RBAC as a method for managing users and permissions based on their role within an organization.
- **Real-World Analogy:** Use the concert analogy to explain different roles (e.g., regular attendee, VIP, backstage crew) and their permissions.
- **Benefits:** Highlight why RBAC is crucial for security, scalability, and managing user access effectively.

Part 2: Components of RBAC

- **Roles:** Explain roles as a collection of permissions.
- **Permissions:** Define permissions as access rights to actions or resources.
- **Users:** Discuss how users are assigned roles, which determine their permissions.

Part 3: Implementing RBAC in Python

- Python example that demonstrates a basic RBAC system with users, roles and permissions.

```
class User:
    def __init__(self, name, role):
        self.name = name
        self.role = role

class Role:
    def __init__(self, name, permissions):
        self.name = name
        self.permissions = permissions

class Permission:
    def __init__(self, name):
        self.name = name

# Create permissions
read_permission = Permission("read")
write_permission = Permission("write")

# Create roles
admin_role = Role("admin", [read_permission, write_permission])
user_role = Role("user", [read_permission])

# Assign roles to users
admin = User("Alice", admin_role)
user = User("Bob", user_role)

# Check if a user has a specific permission
def has_permission(user, permission):
    return permission in user.role.permissions

# Example checks
print(f"Does {admin.name} have write permission? {has_permission(admin, write_permission)}")
print(f"Does {user.name} have write permission? {has_permission(user, read_permission)}")
```

Authorization and OAuth scopes

```
# Assume we have a function to determine the user's role
user_role = get_user_role(user)

# Determine OAuth scopes based on user role
if user_role == "Admin":
    oauth_scopes = ["data:read", "data:write", "profile:read", "profile:write"]
elif user_role == "User":
    oauth_scopes = ["profile:read"]

# OAuth token generation with scopes
oauth_token = generate_oauth_token(user, scopes=oauth_scopes)
```

Best Practices and Advanced Concepts

- Never check if someone has a role but check for permissions only.
- **Dynamic Role Assignment:** Discuss the concept of dynamically assigning roles to users.
- **Scalability:** Talk about designing the RBAC system for scalability as the application grows.
- **Security Considerations:** Emphasize the importance of regular audits and updates to role definitions and permissions.
- Other things to consider:
 - Move roles to the database.
 - Let the user add custom roles.
 - Create groups of permissions that can then also be added to a role.
 - Permissions associated with specific items that you “own”.
 - You could add expiry dates to role assignments, an invitation mechanism, etc.