Restaurant Management System

User Authentication & Authorization - Technical Documentation

Restaurant Management System Development Team

September 4, 2025

Table of Contents



Restaurant Management System

# User Authentication & Authorization System - Technical Documentation

## 1. Overview

This document provides a detailed technical overview of the Restaurant Management System’s authentication and authorization architecture. The system implements a role-based access control (RBAC) system with fine-grained permissions and multi-factor authentication capabilities.

## 2. Database Schema

### 2.1. Core Authentication Tables

#### 2.1.1. Users Table

CREATE TABLE Users (  
 Id INT PRIMARY KEY IDENTITY(1,1),  
 Username NVARCHAR(50) NOT NULL UNIQUE,  
 Email NVARCHAR(100) NOT NULL UNIQUE,  
 PasswordHash NVARCHAR(MAX) NOT NULL,  
 Salt NVARCHAR(MAX) NOT NULL,  
 FirstName NVARCHAR(50) NOT NULL,  
 LastName NVARCHAR(50) NOT NULL,  
 PhoneNumber NVARCHAR(20) NULL,  
 IsActive BIT NOT NULL DEFAULT 1,  
 IsLockedOut BIT NOT NULL DEFAULT 0,  
 FailedLoginAttempts INT NOT NULL DEFAULT 0,  
 LastLoginDate DATETIME NULL,  
 CreatedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 LastModifiedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 MustChangePassword BIT NOT NULL DEFAULT 0,  
 PasswordLastChanged DATETIME NULL,  
 RequiresMFA BIT NOT NULL DEFAULT 0  
);

This table stores user account information with security features: - Password is stored as a hash with a unique salt for each user - Account lockout functionality tracks failed login attempts - MFA requirement flag for enhanced security - Password rotation capability with timestamps

#### 2.1.2. Roles Table

CREATE TABLE Roles (  
 Id INT PRIMARY KEY IDENTITY(1,1),  
 Name NVARCHAR(50) NOT NULL UNIQUE,  
 Description NVARCHAR(200) NULL,  
 IsSystemRole BIT NOT NULL DEFAULT 0,  
 CreatedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 LastModifiedDate DATETIME NOT NULL DEFAULT GETDATE()  
);

Defines all available roles in the system. System roles are protected from deletion or modification.

#### 2.1.3. UserRoles Table (Many-to-Many)

CREATE TABLE UserRoles (  
 UserId INT NOT NULL,  
 RoleId INT NOT NULL,  
 AssignedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 AssignedBy INT NULL,  
 CONSTRAINT PK\_UserRoles PRIMARY KEY (UserId, RoleId),  
 CONSTRAINT FK\_UserRoles\_Users FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE CASCADE,  
 CONSTRAINT FK\_UserRoles\_Roles FOREIGN KEY (RoleId) REFERENCES Roles(Id) ON DELETE CASCADE,  
 CONSTRAINT FK\_UserRoles\_AssignedBy FOREIGN KEY (AssignedBy) REFERENCES Users(Id)  
);

Maps users to their assigned roles with audit information on who assigned the role.

### 2.2. Permission System

#### 2.2.1. Permissions Table

CREATE TABLE Permissions (  
 Id INT PRIMARY KEY IDENTITY(1,1),  
 Name NVARCHAR(100) NOT NULL UNIQUE,  
 Description NVARCHAR(200) NULL,  
 Category NVARCHAR(50) NOT NULL,  
 IsSystemPermission BIT NOT NULL DEFAULT 0  
);

Defines granular permissions that can be assigned to roles.

#### 2.2.2. RolePermissions Table (Many-to-Many)

CREATE TABLE RolePermissions (  
 RoleId INT NOT NULL,  
 PermissionId INT NOT NULL,  
 CONSTRAINT PK\_RolePermissions PRIMARY KEY (RoleId, PermissionId),  
 CONSTRAINT FK\_RolePermissions\_Roles FOREIGN KEY (RoleId) REFERENCES Roles(Id) ON DELETE CASCADE,  
 CONSTRAINT FK\_RolePermissions\_Permissions FOREIGN KEY (PermissionId) REFERENCES Permissions(Id) ON DELETE CASCADE  
);

Maps roles to their assigned permissions.

### 2.3. Multi-Factor Authentication

CREATE TABLE MFAFactors (  
 Id INT PRIMARY KEY IDENTITY(1,1),  
 UserId INT NOT NULL,  
 FactorType NVARCHAR(20) NOT NULL, -- 'Email', 'Phone', 'App'  
 FactorValue NVARCHAR(100) NOT NULL, -- Email, Phone number, or App ID  
 IsVerified BIT NOT NULL DEFAULT 0,  
 IsDefault BIT NOT NULL DEFAULT 0,  
 CreatedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 LastUsedDate DATETIME NULL,  
 CONSTRAINT FK\_MFAFactors\_Users FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE CASCADE  
);

Stores MFA methods configured by users with verification status.

### 2.4. Multi-Outlet Support

CREATE TABLE Outlets (  
 Id INT PRIMARY KEY IDENTITY(1,1),  
 Name NVARCHAR(100) NOT NULL UNIQUE,  
 Location NVARCHAR(200) NULL,  
 IsActive BIT NOT NULL DEFAULT 1,  
 CreatedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 LastModifiedDate DATETIME NOT NULL DEFAULT GETDATE()  
);  
  
CREATE TABLE UserOutletScopes (  
 UserId INT NOT NULL,  
 OutletId INT NOT NULL,  
 AssignedDate DATETIME NOT NULL DEFAULT GETDATE(),  
 AssignedBy INT NULL,  
 CONSTRAINT PK\_UserOutletScopes PRIMARY KEY (UserId, OutletId),  
 CONSTRAINT FK\_UserOutletScopes\_Users FOREIGN KEY (UserId) REFERENCES Users(Id) ON DELETE CASCADE,  
 CONSTRAINT FK\_UserOutletScopes\_Outlets FOREIGN KEY (OutletId) REFERENCES Outlets(Id) ON DELETE CASCADE,  
 CONSTRAINT FK\_UserOutletScopes\_AssignedBy FOREIGN KEY (AssignedBy) REFERENCES Users(Id)  
);

Enables role-based access control across multiple restaurant locations/outlets.

## 3. Authentication Process

### 3.1. Authentication Flow

1. User submits credentials (username/email and password)
2. System verifies credentials against the database
3. If valid, system checks if MFA is required
   * If yes, user is prompted for second factor
   * If no, user is authenticated
4. Upon successful authentication, user’s roles and permissions are loaded
5. A claims-based identity is created and stored in an authentication cookie
6. User is redirected to the requested resource or default page

### 3.2. Key Authentication Stored Procedures

#### 3.2.1. sp\_AuthenticateUser

-- Stored procedure for user authentication  
CREATE PROCEDURE [dbo].[sp\_AuthenticateUser]  
 @Username NVARCHAR(50),  
 @Password NVARCHAR(100),  
 @Success BIT OUTPUT,  
 @Message NVARCHAR(200) OUTPUT,  
 @UserId INT OUTPUT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Initialize output parameters  
 SET @Success = 0;  
 SET @Message = '';  
 SET @UserId = NULL;  
   
 -- Check if user exists  
 DECLARE @StoredHash NVARCHAR(MAX);  
 DECLARE @StoredSalt NVARCHAR(MAX);  
 DECLARE @IsActive BIT;  
 DECLARE @IsLockedOut BIT;  
 DECLARE @FailedAttempts INT;  
   
 SELECT   
 @UserId = Id,  
 @StoredHash = PasswordHash,  
 @StoredSalt = Salt,  
 @IsActive = IsActive,  
 @IsLockedOut = IsLockedOut,  
 @FailedAttempts = FailedLoginAttempts  
 FROM   
 Users   
 WHERE   
 Username = @Username;  
   
 -- Check if user exists  
 IF @UserId IS NULL  
 BEGIN  
 SET @Message = 'Invalid username or password';  
 RETURN;  
 END  
   
 -- Check if account is active  
 IF @IsActive = 0  
 BEGIN  
 SET @Message = 'Account is disabled';  
 RETURN;  
 END  
   
 -- Check if account is locked out  
 IF @IsLockedOut = 1  
 BEGIN  
 SET @Message = 'Account is locked due to too many failed login attempts';  
 RETURN;  
 END  
   
 -- Verify password hash (simplified for documentation)  
 -- In actual implementation, this would use .NET code to verify the hash  
   
 -- Update login statistics  
 UPDATE Users SET   
 LastLoginDate = GETDATE(),  
 FailedLoginAttempts = 0  
 WHERE   
 Id = @UserId;  
   
 SET @Success = 1;  
 SET @Message = 'Authentication successful';  
END

#### 3.2.2. sp\_GetUserRolesAndPermissions

CREATE PROCEDURE [dbo].[sp\_GetUserRolesAndPermissions]  
 @UserId INT  
AS  
BEGIN  
 SET NOCOUNT ON;  
   
 -- Get user roles  
 SELECT   
 r.Id,  
 r.Name,  
 r.Description,  
 r.IsSystemRole  
 FROM   
 Roles r  
 INNER JOIN   
 UserRoles ur ON r.Id = ur.RoleId  
 WHERE   
 ur.UserId = @UserId;  
   
 -- Get user permissions  
 SELECT   
 p.Id,  
 p.Name,  
 p.Description,  
 p.Category,  
 p.IsSystemPermission  
 FROM   
 Permissions p  
 INNER JOIN   
 RolePermissions rp ON p.Id = rp.PermissionId  
 INNER JOIN   
 UserRoles ur ON rp.RoleId = ur.RoleId  
 WHERE   
 ur.UserId = @UserId;  
   
 -- Get user outlet scopes  
 SELECT  
 o.Id,  
 o.Name,  
 o.Location,  
 o.IsActive  
 FROM  
 Outlets o  
 INNER JOIN  
 UserOutletScopes uos ON o.Id = uos.OutletId  
 WHERE  
 uos.UserId = @UserId;  
END

## 4. Role-Based Access Control (RBAC)

### 4.1. User Role Enumeration

The system defines the following standard user roles:

public enum UserRole  
{  
 Guest, // Guest/Customer  
 Host, // Manages seating and reservations  
 Server, // Waiter/Waitress who takes orders  
 Cashier, // Handles payments  
 StationChef, // Chef responsible for specific station  
 Expeditor, // Coordinates food delivery from kitchen to servers  
 InventoryClerk, // Manages inventory  
 PurchasingManager, // Handles purchasing of supplies  
 RestaurantManager, // Overall management  
 DeliveryRider, // Delivery personnel  
 Accountant, // Handles finances  
 SystemAdmin, // System administrator  
   
 // Integration roles:  
 CRMMarketing, // Marketing system  
 PaymentGateway, // Payment processing  
 Aggregator, // Third-party order aggregator  
 ERPAccounting, // ERP/Accounting system  
 MessagingProvider, // SMS/WhatsApp provider  
 BIAnalytics // Business Intelligence/Analytics  
}

### 4.2. Permission Categories

Permissions are grouped into functional categories:

1. **Menu Management** - Managing menu items, categories, and pricing
2. **Order Processing** - Taking, updating, and canceling orders
3. **Kitchen Operations** - Viewing and updating order status, recipe information
4. **Inventory Management** - Stock control, ingredient tracking
5. **Reservation System** - Table management, booking control
6. **Financial Operations** - Payments, refunds, accounting functions
7. **User Administration** - Creating and managing user accounts
8. **System Configuration** - Global system settings

### 4.3. Permission Assignment

Each role is granted specific permissions based on job responsibilities. For example:

-- Assign menu view permission to Server role  
INSERT INTO RolePermissions (RoleId, PermissionId)  
SELECT   
 (SELECT Id FROM Roles WHERE Name = 'Server'),  
 (SELECT Id FROM Permissions WHERE Name = 'Menu.View');  
  
-- Assign menu edit permission to RestaurantManager role  
INSERT INTO RolePermissions (RoleId, PermissionId)  
SELECT   
 (SELECT Id FROM Roles WHERE Name = 'RestaurantManager'),  
 (SELECT Id FROM Permissions WHERE Name = 'Menu.Edit');

## 5. Authorization Implementation

### 5.1. Claims-Based Authorization

The system implements ASP.NET Core’s claims-based identity system:

// Populating user claims during sign-in  
public async Task SignInUserAsync(AuthUser user, bool rememberMe)  
{  
 var claims = new List<Claim>  
 {  
 new Claim(ClaimTypes.NameIdentifier, user.Id.ToString()),  
 new Claim(ClaimTypes.Name, user.Username),  
 new Claim(ClaimTypes.Email, user.Email),  
 new Claim(ClaimTypes.GivenName, user.FirstName),  
 new Claim(ClaimTypes.Surname, user.LastName)  
 };  
   
 // Add roles as claims  
 foreach (var role in user.Roles)  
 {  
 claims.Add(new Claim(ClaimTypes.Role, role.Name));  
 }  
   
 // Add permissions as claims  
 foreach (var permission in user.Permissions)  
 {  
 claims.Add(new Claim("Permission", permission));  
 }  
   
 var claimsIdentity = new ClaimsIdentity(  
 claims, CookieAuthenticationDefaults.AuthenticationScheme);  
   
 var authProperties = new AuthenticationProperties  
 {  
 IsPersistent = rememberMe,  
 ExpiresUtc = DateTimeOffset.UtcNow.AddDays(rememberMe ? 30 : 1)  
 };  
   
 await \_httpContextAccessor.HttpContext.SignInAsync(  
 CookieAuthenticationDefaults.AuthenticationScheme,   
 new ClaimsPrincipal(claimsIdentity),  
 authProperties);  
}

### 5.2. Controller Authorization

Controllers and actions are secured using the [Authorize] attribute with role or policy requirements:

// Role-based authorization  
[Authorize(Roles = "RestaurantManager,SystemAdmin")]  
public class MenuManagementController : Controller  
{  
 [HttpPost]  
 public IActionResult Create(MenuItem model) { ... }  
}  
  
// Policy-based authorization for fine-grained control  
[Authorize(Policy = "RequireMenuEditPermission")]  
public IActionResult Edit(int id) { ... }

### 5.3. Policy-Based Authorization

Custom authorization policies are defined in the application startup:

builder.Services.AddAuthorization(options =>  
{  
 // Permission-based policies  
 options.AddPolicy("RequireMenuEditPermission", policy =>  
 policy.RequireClaim("Permission", "Menu.Edit"));  
   
 options.AddPolicy("RequireUserAdministration", policy =>  
 policy.RequireClaim("Permission", "User.Create", "User.Edit", "User.Delete"));  
   
 // Multi-role policies  
 options.AddPolicy("RequireKitchenAccess", policy =>  
 policy.RequireRole("StationChef", "Expeditor", "RestaurantManager", "SystemAdmin"));  
   
 // Combined policies  
 options.AddPolicy("RequireFinancialAccess", policy =>  
 policy.RequireRole("Accountant", "RestaurantManager", "SystemAdmin")  
 .RequireClaim("Permission", "Financial.View"));  
});

### 5.4. View-Level Authorization

Authorization is also enforced in views:

@if (User.IsInRole("RestaurantManager") || User.IsInRole("SystemAdmin"))  
{  
 <a asp-controller="Menu" asp-action="Create" class="btn btn-primary">Create New Menu Item</a>  
}  
  
@if (User.HasClaim("Permission", "Financial.ViewReports"))  
{  
 <div class="card">  
 <div class="card-header">Financial Reports</div>  
 <div class="card-body">  
 <!-- Financial reporting UI -->  
 </div>  
 </div>  
}

## 6. Security Features

### 6.1. Password Security

* Passwords are stored using industry-standard hashing (PBKDF2 with high iteration count)
* Each user has a unique salt
* Password complexity requirements enforced:
  + Minimum 8 characters
  + Requires uppercase, lowercase, number, and special character
  + Password history prevents reuse of previous passwords
  + Configurable password expiration

### 6.2. Multi-Factor Authentication

* Optional MFA can be required for specific roles
* Supported methods:
  + Email verification codes
  + SMS verification codes
  + Time-based one-time password (TOTP) authenticator apps

### 6.3. Account Lockout

* Accounts are automatically locked after a configurable number of failed login attempts
* Locked accounts require administrator intervention to unlock
* All lockout events are logged for audit purposes

### 6.4. Emergency Access

The system includes an emergency backdoor mechanism for system administrators:

// EMERGENCY BACKDOOR: Accept admin/Admin@123 unconditionally to ensure access  
if (username.ToLower() == "admin" && password == "Admin@123")  
{  
 // Create an admin user with full permissions for emergency access  
 var adminUser = new AuthUser  
 {  
 Id = 1,  
 Username = "admin",  
 Email = "admin@restaurant.com",  
 FirstName = "System",  
 LastName = "Administrator",  
 RequiresMFA = false,  
 Roles = new List<AuthUserRole> { new AuthUserRole { Id = 1, Name = "Administrator" } },  
 Permissions = new List<string> { "All" },  
 Outlets = new List<Outlet>()  
 };  
   
 return (true, "Authentication successful", adminUser);  
}

**Important:** This backdoor should be removed or properly secured in a production environment.

## 7. Data Access Layer

### 7.1. User Data Model

public class User  
{  
 public int Id { get; set; }  
  
 [Required]  
 [StringLength(50)]  
 public string Username { get; set; }  
  
 [Required]  
 [StringLength(100)]  
 public string Password { get; set; } // Stored as hash  
  
 [Required]  
 [StringLength(50)]  
 public string FirstName { get; set; }  
  
 [StringLength(50)]  
 public string LastName { get; set; }  
   
 public string FullName   
 {  
 get { return $"{FirstName} {LastName}".Trim(); }  
 }  
  
 [EmailAddress]  
 [StringLength(100)]  
 public string Email { get; set; }  
  
 [Phone]  
 [StringLength(20)]  
 public string Phone { get; set; }  
  
 [Required]  
 public UserRole Role { get; set; }  
  
 public bool IsActive { get; set; } = true;  
   
 public DateTime CreatedAt { get; set; } = DateTime.Now;  
   
 public DateTime? LastLogin { get; set; }  
}

### 7.2. Authentication Models

public class AuthUser  
{  
 public int Id { get; set; }  
   
 [Required]  
 [StringLength(50)]  
 public string Username { get; set; }  
   
 [Required]  
 [EmailAddress]  
 [StringLength(100)]  
 public string Email { get; set; }  
   
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string FullName => $"{FirstName} {LastName}";  
   
 public bool RequiresMFA { get; set; }  
   
 public List<AuthUserRole> Roles { get; set; } = new List<AuthUserRole>();  
 public List<string> Permissions { get; set; } = new List<string>();  
 public List<Outlet> Outlets { get; set; } = new List<Outlet>();  
}

## 8. Best Practices & Security Considerations

### 8.1. Security Best Practices

1. **Regular Password Rotation** - Enforce password changes every 60-90 days
2. **Audit Logging** - Track all authentication and authorization events
3. **Session Management** - Sessions time out after 30 minutes of inactivity
4. **Transport Security** - All communications secured via HTTPS
5. **Error Handling** - Generic error messages for failed logins
6. **Input Validation** - All user inputs validated server-side

### 8.2. Application Architecture Recommendations

1. **Service-Based Pattern** - Authentication logic is encapsulated in dedicated services
2. **Repository Pattern** - Data access is abstracted through repositories
3. **Dependency Injection** - Services are injected where needed
4. **Claims Transformation** - Claims are loaded dynamically to ensure current permissions

### 8.3. Known Issues and Limitations

1. ⚠️ Emergency backdoor in AuthService should be removed for production
2. ⚠️ The current implementation has incomplete MFA verification
3. ⚠️ Password history implementation is not fully implemented

## 9. Testing Approach

### 9.1. Unit Testing

// Sample unit test for AuthService  
[Test]  
public async Task AuthenticateUser\_WithValidCredentials\_ReturnsSuccess()  
{  
 // Arrange  
 var mockHttpContextAccessor = new Mock<IHttpContextAccessor>();  
 var mockConfiguration = new Mock<IConfiguration>();  
 mockConfiguration.Setup(c => c.GetConnectionString("DefaultConnection"))  
 .Returns("Test Connection String");  
   
 var authService = new AuthService(mockConfiguration.Object, mockHttpContextAccessor.Object);  
   
 // Act  
 var result = await authService.AuthenticateUserAsync("validuser", "ValidPassword123!");  
   
 // Assert  
 Assert.IsTrue(result.success);  
 Assert.IsNotNull(result.user);  
 Assert.AreEqual("validuser", result.user.Username);  
}

### 9.2. Integration Testing

Integration tests verify that the authentication system works correctly with the database:

1. User creation and retrieval
2. Role assignment and verification
3. Permission checking
4. Login flows
5. Password reset functionality

## 10. Deployment Considerations

### 10.1. Environment Configuration

Different authentication settings for various environments:

* **Development** - Relaxed security, no MFA
* **Testing** - Mimics production, with test accounts
* **Production** - Full security measures active

### 10.2. Database Migration Scripts

Proper migration scripts ensure the authentication tables are created correctly across all environments.

### 10.3. Secret Management

Sensitive authentication configuration is stored in Azure Key Vault or similar secure storage.