Authorization System Guide

Overview

The IKEA DocuScan application uses a **Hybrid Authorization Approach** that combines: 1. **Claims-based Authorization** - For basic, lightweight checks (HasAccess, IsSuperUser) 2. **Active Directory Group Roles** - Role claims based on AD group membership (Reader, Publisher, SuperUser) 3. **CurrentUserService** - For detailed permission filtering (document types, counter parties, countries)

This approach provides both performance (claims are cached in the authentication ticket) and flexibility (detailed permissions loaded on-demand).

Architecture

1. Database Schema

DocuScanUser Table: - `UserId` (PK) - Unique user identifier - `AccountName` - Windows domain - `UserIdentifier` - Alternative identifier - `IsSuperUser` - Flag for admin privileges - `LastLogon` - Timestamp of last login - `CreatedOn`, `ModifiedOn` - Audit timestamps

UserPermissions Table: - `PermissionId` (PK) - `UserId` (FK to DocuScanUser) - `DocumentTypeId` - Allowed document type (nullable = all) - `CounterPartyId` - Allowed counter party (nullable = all) - `CountryCode` - Allowed country (nullable = all)

2. Authentication Flow

```
User Request
    ↓
Windows Authentication (or Test Auth in dev)
    ↓
WindowsIdentityMiddleware
    ↓
Load User from DocuScanUser table
    ↓
Load UserPermissions
    ↓
Check AD Group Membership (Reader, Publisher, SuperUser)
    ↓
Add Claims to ClaimsPrincipal:
    - "UserId"
    - "IsSuperUser" (from database OR AD group)
    - "HasAccess"
    - ClaimTypes.Role = "Reader" (if in AD group)
    - ClaimTypes.Role = "Publisher" (if in AD group)
    - ClaimTypes.Role = "SuperUser" (if in database OR AD group)
    ↓
Request continues with authenticated user
```

3. Authorization Components

Claims (Added by WindowsIdentityMiddleware): - `UserId` - Database user ID - `IsSuperUser` - “true” or “false” (from database OR AD group) - `HasAccess` - “true” or “false” - `ClaimTypes.Role` - One or more of: - “Reader” (if user is in ADGroup.Builtin.Reader) - “Publisher” (if user is in ADGroup.Builtin.Publisher) - “SuperUser” (if user is in database OR ADGroup.Builtin.SuperUser)

Authorization Policies (in Program.cs): - `HasAccess` - Requires HasAccess claim = true

- `SuperUser` - Requires IsSuperUser claim = true - Default policy - Requires authenticated user

CurrentUserService (Scoped): - Loads once per request and caches - Provides detailed permission checking - Methods: `GetCurrentUserAsync()`, `CanAccessDocument()`, etc.

Active Directory Group Roles

Overview

The application supports three built-in AD groups that are automatically mapped to role claims:

1. **Reader** (`ADGroup.Builtin.Reader`)
   - Read-only access to documents
   - Can view and search documents
   - Cannot modify, register, or delete documents
2. **Publisher** (`ADGroup.Builtin.Publisher`)
   - Full document management access
   - Can create, edit, and register documents
   - Can send links and attachments
   - Cannot delete documents or manage users
3. **SuperUser** (`ADGroup.Builtin.SuperUser`)
   - Full administrative access
   - Can perform all operations including delete
   - Can manage user permissions
   - Bypasses all permission filters

Configuration

AD groups are configured in `appsettings.json`:

```
{
  "IkeaDocuScan": {
    "ADGroupReader": "ADGroup.Builtin.Reader",
    "ADGroupPublisher": "ADGroup.Builtin.Publisher",
    "ADGroupSuperUser": "ADGroup.Builtin.SuperUser"
  }
}
```

Important Notes: - Group names can be configured to match your organization’s AD structure - You can use simple group names (e.g., “Readers”) or domain-qualified names (e.g., “DOMAIN\DocuScan_Readers”) - If a user is in the SuperUser AD group, they automatically get `IsSuperUser=true` and `HasAccess=true`, overriding database values - Users can be in multiple AD groups and will receive all corresponding role claims

How AD Groups Work with Database Permissions

The system uses **BOTH** AD groups and database permissions:

1. **AD Groups provide role-based access** (Reader, Publisher, SuperUser)
2. **Database permissions provide filtering** (by document type, country, counter party)

Example Scenarios:

- **User in “Reader” AD group + Database permissions for CountryCode=“US”**
  - Can read documents
  - Only sees documents where CountryCode = “US”
- **User in “Publisher” AD group + Database SuperUser flag = true**
  - Can publish/edit documents
  - Sees ALL documents (SuperUser overrides filters)
- **User in “SuperUser” AD group (no database record)**

  - Full admin access
  - Sees ALL documents
  - Can perform all operations

Usage Examples

1. Basic Authorization - Using Claims

Claims-based checks are fast and suitable for page-level authorization.

Razor Component Authorization

```
@page "/documents"
@attribute [Authorize(Policy = "HasAccess")]

<h3>Documents</h3>
```

Check if Super User in Component

```
@inject AuthenticationStateProvider AuthStateProvider

@code {
    private bool isSuperUser = false;

    protected override async Task OnInitializedAsync()
    {
        var authState = await
AuthStateProvider.GetAuthenticationStateAsync();
        var superUserClaim = authState.User.FindFirst("IsSuperUser");
        isSuperUser = superUserClaim != null
            && bool.TryParse(superUserClaim.Value, out bool isSuper)
            && isSuper;
    }
}
```

Conditional Rendering Based on Role

```
<!-- Using Policy -->
<AuthorizeView Policy="SuperUser">
    <Authorized>
        <button class="btn btn-danger" @onclick="DeleteAll">Delete
All</button>
    </Authorized>
    <NotAuthorized>
        <p>Admin access required</p>
    </NotAuthorized>
</AuthorizeView>

<!-- Using AD Group Roles -->
<AuthorizeView Roles="Reader">
    <Authorized>
        <p>You have read-only access</p>
    </Authorized>
</AuthorizeView>

<AuthorizeView Roles="Publisher">
    <Authorized>
        <button class="btn btn-primary"
@onclick="PublishDocument">Publish</button>
    </Authorized>
</AuthorizeView>

<AuthorizeView Roles="SuperUser">
    <Authorized>
        <button class="btn btn-danger"
@onclick="DeleteDocument">Delete</button>
    </Authorized>
</AuthorizeView>

<!-- Multiple roles (user must be in at least one) -->
<AuthorizeView Roles="Publisher,SuperUser">
    <Authorized>
        <button @onclick="EditDocument">Edit Document</button>
    </Authorized>
</AuthorizeView>
```

Check AD Group Membership in Code

```
@inject AuthenticationStateProvider AuthStateProvider

@code {
    private bool isReader = false;
    private bool isPublisher = false;
    private bool isSuperUser = false;

    protected override async Task OnInitializedAsync()
    {
        var authState = await AuthStateProvider.GetAuthenticationStateAsyn
        var user = authState.User;

        isReader = user.IsInRole("Reader");
        isPublisher = user.IsInRole("Publisher");
        isSuperUser = user.IsInRole("SuperUser");
    }
}
```

2. Detailed Authorization - Using CurrentUserService

For filtering data based on document types, counter parties, or countries.

Check Document Access

```
@inject ICurrentUserService CurrentUserService

@code {
    private async Task<bool> CanUserViewDocument(Document doc)
    {
        var currentUser = await CurrentUserService.GetCurrentUserAsync();

        return currentUser.CanAccessDocument(
            doc.DocumentTypeId,
            doc.CounterPartyId,
            doc.CountryCode
        );
    }
}
```

Filter Document List

```
@inject ICurrentUserService CurrentUserService
@inject IDocumentService DocumentService

@code {
    private List<DocumentDto> filteredDocuments = new();

    protected override async Task OnInitializedAsync()
    {
        var currentUser = await CurrentUserService.GetCurrentUserAsync();
        var allDocuments = await DocumentService.GetAllDocumentsAsync();

        if (currentUser.IsSuperUser)
        {
            // Super user sees everything
            filteredDocuments = allDocuments;
        }
        else
        {
            // Filter based on user permissions
            filteredDocuments = allDocuments
                .Where(doc => currentUser.CanAccessDocument(
                    doc.DocumentTypeId,
                    doc.CounterPartyId,
                    doc.CountryCode))
                .ToList();
        }
    }
}
```

Synchronous Property Access

```
@inject ICurrentUserService CurrentUserService

@code {
    protected override void OnInitialized()
    {
        // Note: These properties return cached values
        // You must call GetCurrentUserAsync() first to load the user

        if (CurrentUserService.HasAccess)
        {
            // User has system access
        }

        if (CurrentUserService.IsSuperUser)
        {
            // User is super user
        }

        var userId = CurrentUserService.UserId;
    }
}
```

3. Access Request Flow

Handling Unauthorized Users

```
@page "/"
@inject ICurrentUserService CurrentUserService
@inject NavigationManager Navigation

@code {
    protected override async Task OnInitializedAsync()
    {
        var currentUser = await CurrentUserService.GetCurrentUserAsync();

        if (!currentUser.HasAccess)
        {
            Navigation.NavigateTo("/access-denied");
        }
    }
}
```

Custom Access Request

```
@inject ICurrentUserService CurrentUserService

<button @onclick="RequestAccess">Request Access</button>

@code {
    private async Task RequestAccess()
    {
        var username = "DOMAIN\\username";
        var reason = "Need access for document processing";

        var result = await CurrentUserService.RequestAccessAsync(username,
reason);

        if (result.Success)
        {
            // Show success message
        }
    }
}
```

4. Service-Level Authorization

In a Service Class

```
public class CustomService
{
    private readonly ICurrentUserService _currentUserService;

    public CustomService(ICurrentUserService currentUserService)
    {
        _currentUserService = currentUserService;
    }

    public async Task<List<Invoice>> GetUserInvoicesAsync()
    {
        var currentUser = await _currentUserService.GetCurrentUserAsync();

        if (!currentUser.HasAccess)
        {
            throw new UnauthorizedAccessException("User does not have acce
        }

        // Load invoices and filter
        var invoices = await LoadInvoicesFromDatabase();

        if (currentUser.IsSuperUser)
        {
            return invoices; // Super user sees all
        }

        // Filter by user permissions
        return invoices.Where(inv =>
            currentUser.CanAccessCountry(inv.CountryCode) &&
            currentUser.CanAccessCounterParty(inv.CounterPartyId)
        ).ToList();
    }
}
```

5. API Endpoint Authorization

Minimal API with Authorization

```
public static class DocumentEndpoints
{
    public static void MapDocumentEndpoints(this WebApplication app)
    {
        var group = app.MapGroup("/api/documents")
            .RequireAuthorization("HasAccess");

        // All authenticated users with access can read
        group.MapGet("/", GetDocuments);

        // Only Publishers and SuperUsers can create
        group.MapPost("/", CreateDocument)
            .RequireAuthorization(policy => policy.RequireRole("Publisher"

        // Only Publishers and SuperUsers can edit
        group.MapPut("/{id}", UpdateDocument)
            .RequireAuthorization(policy => policy.RequireRole("Publisher"

        // Only SuperUsers can delete
        group.MapDelete("/{id}", DeleteDocument)
            .RequireAuthorization(policy => policy.RequireRole("SuperUser"
    }

    private static async Task<IResult> GetDocuments(
        ICurrentUserService currentUserService,
        IDocumentService documentService)
    {
        var currentUser = await currentUserService.GetCurrentUserAsync();
        var documents = await documentService.GetAllDocumentsAsync();

        // Filter based on permissions
        var filtered = documents.Where(doc =>
            currentUser.CanAccessDocument(
                doc.DocumentTypeId,
                doc.CounterPartyId,
                doc.CountryCode)
        );

        return Results.Ok(filtered);
    }

    private static async Task<IResult> DeleteDocument(
        int id,
        IDocumentService documentService)
    {
        // Only super users can reach this endpoint
        await documentService.DeleteDocumentAsync(id);
        return Results.Ok();
    }
}
```

Best Practices

When to Use AD Roles vs Claims vs CurrentUserService

Use AD Role Claims (e.g., Reader, Publisher, SuperUser) when: - □ Operation-level authorization (can user perform this action?) - □ Simple, organization-wide role checks - □ UI buttons/features that vary by role - □ API endpoint authorization by operation type - □ Performance is critical (claims are cached) - □ Want to leverage existing AD group structure

Use Claims (Policy-based, e.g., HasAccess, IsSuperUser) when: - □ Page-level authorization (who can access this page?) - □ Quick boolean checks (does user have any access?) - □ Fallback authorization when AD groups aren't available - □ Performance is critical (claims are cached)

Use CurrentUserService when: - □ Filtering data by document type, country, or counter party - □ Row-level authorization (which documents can user see?) - □ Complex permission logic combining multiple attributes - □ Need detailed permission information - □ Need to check specific combinations of permissions

Typical Usage Pattern: 1. AD Role for "Can they do this?" → `@attribute [Authorize(Roles = "Publisher")]` 2. CurrentUserService for "Which documents?" → Filter by document type/country/party

Performance Tips

1. **Load CurrentUser Once Per Request**

```
// DO THIS (loads once and caches)
var currentUser = await CurrentUserService.GetCurrentUserAsync();
foreach (var doc in documents)
{
    if (currentUser.CanAccessDocument(...)) { }
}

// DON'T DO THIS (loads on every iteration)
foreach (var doc in documents)
{
    var currentUser = await CurrentUserService.GetCurrentUserAsync();
}
```

2. **Use Synchronous Properties When Possible**

```
// After loading once
await CurrentUserService.GetCurrentUserAsync();

// Use synchronous properties
if (CurrentUserService.IsSuperUser) { }
if (CurrentUserService.HasAccess) { }
```

3. **Prefer Claims for Simple Checks**

```
// FASTER (uses cached claims)
var isSuperUserClaim = User.FindFirst("IsSuperUser");

// SLOWER (database query)
var currentUser = await CurrentUserService.GetCurrentUserAsync();
var isSuperUser = currentUser.IsSuperUser;
```

Security Considerations

1. **Always check permissions server-side**
   - Never trust client-side checks alone
   - Use `[Authorize]` attributes on pages and API endpoints
2. **Filter data at the service layer**
   - Don't rely on UI hiding data
   - Filter in the service before returning to the component
3. **Invalidate cache when permissions change**

```
await UpdateUserPermissions(userId);
CurrentUserService.InvalidateCache();
```

4. **Log authorization failures**
   - WindowsIdentityMiddleware logs access denials
   - Authorization handlers log policy failures

Configuration

appsettings.json

```
{
  "IkeaDocuScan": {
    "ContactEmail": "docuscan-admin@company.com",
    "ADGroupReader": "ADGroup.Builtin.Reader",
    "ADGroupPublisher": "ADGroup.Builtin.Publisher",
    "ADGroupSuperUser": "ADGroup.Builtin.SuperUser"
  }
}
```

Customizing AD Group Names:

You can configure the AD group names to match your organization's structure:

```
{
  "IkeaDocuScan": {
    "ADGroupReader": "IKEA\\DocuScan_Readers",
    "ADGroupPublisher": "IKEA\\DocuScan_Publishers",
    "ADGroupSuperUser": "IKEA\\DocuScan_Admins"
  }
}
```

Notes: - Use double backslashes (`\\`) in JSON for domain-qualified group names - Group names are case-insensitive - Set to `null` or empty string to disable a specific role - Changes require application restart to take effect

Environment-Specific Configuration

For production, use encrypted configuration:

```
# Run the ConfigEncryptionTool
cd ConfigEncryptionTool
dotnet run
```

This creates `secrets.encrypted.json` with DPAPI-encrypted values.

Testing

Test AD Group Membership

Check if user is in an AD group:

```
# PowerShell command to check AD group membership
Get-ADUser -Identity "username" -Properties MemberOf |
    Select-Object -ExpandProperty MemberOf |
    Where-Object { $_ -like "*DocuScan*" }
```

Add user to AD group (requires AD admin privileges):

```
# Add user to Reader group
Add-ADGroupMember -Identity "ADGroup.Builtin.Reader" -Members "username"

# Add user to Publisher group
Add-ADGroupMember -Identity "ADGroup.Builtin.Publisher" -Members "username

# Add user to SuperUser group
Add-ADGroupMember -Identity "ADGroup.Builtin.SuperUser" -Members "username
```

Note: AD group changes may take a few minutes to propagate. User may need to log out and log back in.

Test Super User

```sql
-- In database, set IsSuperUser = true
UPDATE DocuScanUser
SET IsSuperUser = 1
WHERE AccountName = 'DOMAIN\testuser'
```

OR add user to SuperUser AD group (preferred method):

```powershell
Add-ADGroupMember -Identity "ADGroup.Builtin.SuperUser" -Members "testuser
```

Test Permissions

```sql
-- Grant specific permissions
INSERT INTO UserPermissions (UserId, DocumentTypeId, CounterPartyId, Count
VALUES (1, 5, NULL, 'US')  -- User 1 can access DocumentType 5 in US
```

Test Access Request

1. Remove user from DocuScanUser table
2. Navigate to application
3. Should be redirected to /access-denied
4. Submit access request
5. Check DocuScanUser table for new record

Troubleshooting

User Not Authenticated

Symptom: User is not authenticated, gets redirected to access denied

Check: 1. Windows Authentication is configured in IIS 2. WindowsIdentityMiddleware is registered in Program.cs 3. User exists in Active Directory

User Has No Access

Symptom: Authenticated but HasAccess = false

Check: 1. User exists in DocuScanUser table 2. User has at least one permission in UserPermissions table 3. Or user has IsSuperUser = true

Permissions Not Applied

Symptom: User sees documents they shouldn’t

Check: 1. CurrentUserService is being used to filter data 2. Service is scoped (not singleton) 3. Cache hasn’t become stale (call InvalidateCache if needed)

Performance Issues

Symptom: Slow page loads, too many database queries

Check: 1. CurrentUserService.GetCurrentUserAsync() called only once per request 2. Using AsNoTracking() on permission queries 3. Consider adding database indexes on UserPermissions foreign keys

AD Group Roles Not Working

Symptom: User is in AD group but role claims are not added

Check: 1. Verify AD group configuration in appsettings.json matches actual AD group names 2. Check application logs for AD group membership errors in WindowsIdentityMiddleware 3. Verify user is actually in the AD group: `powershell Get-ADUser -Identity "username" -Properties MemberOf | Select-Object -ExpandProperty MemberOf` 4. User may need to log out and log back in after being added to AD group 5. On Linux/WSL development, TestAuthenticationHandler doesn't check AD groups (Windows only)

Symptom: IsInRole() returns false even though user is in group

Check: 1. Group name format - try both "GroupName" and "DOMAIN\GroupName" 2. Check if Windows Authentication is properly configured (not using TestAuthenticationHandler) 3. Application pool identity in IIS must have permission to query AD groups

Symptom: SuperUser AD group not overriding database permissions

Check: 1. Verify ADGroupSuperUser is configured in appsettings.json 2. Check logs to confirm user was detected in SuperUser AD group 3. Ensure WindowsIdentityMiddleware is registered AFTER UseAuthentication() in Program.cs

---

File Reference

- **WindowsIdentityMiddleware.cs**: `/IkeaDocuScan-Web/Middleware/WindowsIdentityMiddleware.cs:19`
- **CurrentUserService.cs**: `/IkeaDocuScan-Web/Services/CurrentUserService.cs:33`
- **ICurrentUserService.cs**: `/IkeaDocuScan.Shared/Interfaces/ICurrentUserService.cs:12`
- **CurrentUser.cs**: `/IkeaDocuScan.Shared/Models/Authorization/CurrentUser.cs:32`
- **UserAccessHandler.cs**: `/IkeaDocuScan-Web/Authorization/UserAccessHandler.cs:12`
- **AccessDenied.razor**: `/IkeaDocuScan-Web.Client/Pages/AccessDenied.razor:1`
- **Program.cs**: `/IkeaDocuScan-Web/Program.cs:48` (Authorization setup)

Migration from Previous System

If migrating from a system with AccountName-based permissions:

1. Run database migration scripts (04-07)
2. User records will be automatically created from existing permissions
3. Set IsSuperUser flag for admin users manually
4. Update code to use new authorization system
5. Test thoroughly with different permission combinations

Future Enhancements

Potential improvements to consider:

1. **Email Notifications**: Send email to admin when access is requested
2. **Permission Groups**: Create groups of permissions for easier management
3. **Temporary Permissions**: Time-limited access grants
4. **Audit Log Integration**: Track all permission checks
5. **Admin UI**: Web interface for managing user permissions
6. **LDAP Group Sync**: Auto-assign permissions based on AD groups