Filter...

latest (v7.1)

ABP Framework

Overall

Introduction
Tutorials & Articles
NLayer Architecture
Module System
Startup Configuration
Multi-Tenancy
OWIN Integration
Debugging
API Reference

Common Structures

Dependency Injection
Session
Caching
Logging
Setting Management
Timing
Object To Object Mapping (and AutoMapper Integration)
Email Sending (and MailKit Integration)

Domain Layer

Entities
Multi-Lingual Entities
Value Objects
Repositories
Domain Services
Specifications
Unit Of Work
Domain Events (EventBus)
Data Filters
Dynamic Parameter System
Object Comparators

Application Layer

Application Services
Data Transfer Objects
Validating Data Transfer Objects

In this document

Introduction
About IPermissionChecker
Defining Permissions
Checking Permissions
Using AbpAuthorize Attribute
Using IPermissionChecker
In Razor Views
Client Side (JavaScript)
Permission Manager

Edit on GitHub

Introduction

Almost all enterprise applications use authorization at some level. Authorization is used to check if a user is allowed to perform some specific operation in the application. ASP.NET Boilerplate defines a **permission based** infrastructure to implement authorization.

About IPermissionChecker

The Authorization system uses **IPermissionChecker** to check permissions. While you can implement it in your own way, it's fully implemented in the **Module Zero** project. If it's not implemented, NullPermissionChecker is used which grants all permissions to everyone.

Defining Permissions

A unique **permission** is defined for each operation that needs to be authorized. We need to define a permission before it is used. ASP.NET Boilerplate is designed to be modular, so different modules can have different permissions. A module should create a class derived from **AuthorizationProvider** in order to define it's permissions. An example authorization provider is shown below:

Copy

```
public class MyAuthorizationProvider : AuthorizationProvider
{
    public override void SetPermissions(IPermissionDefinitionContext context)
    {
        var administration = context.CreatePermission("Administration");

        var userManagement =
administration.CreateChildPermission("Administration.UserManagement");
        userManagement.CreateChildPermission("Administration.UserManagement.CreateUser");

        var roleManagement =
administration.CreateChildPermission("Administration.RoleManagement");
    }
}
```

IPermissionDefinitionContext has methods to get and create permissions.

A permission is defined with these properties:

- **Name**: a system-wide **unique** name. It's a good idea to define a const string for a permission name instead of a magic string. We prefer to use . (dot) notation for hierarchical names but it's not required. You can set any name you like. The only rule is that it must be unique.
- **Display name**: A localizable string that can be used to show the permission later in UI.
- **Description**: A localizable string that can be used to show the definition of the permission later in UI.
- **MultiTenancySides**: For the multi-tenant application, a permission can be used by tenants or the host. This is a **Flags** enumeration and thus a permission can be used on both sides.
- **featureDependency**: Can be used to declare a dependency to features. Thus, this permission can be granted only if the feature dependency is satisfied. It waits for an object implementing IFeatureDependency. The default implementation is the SimpleFeatureDependency class. Example usage: `new SimpleFeatureDependency("MyFeatureName")`

Permissions can have parent and child permissions. While this does not affect permission checking, it helps to group the permissions in the UI.

After creating an authorization provider, we should register it in the PreInitialize method of our module:

Copy

```
Configuration.Authorization.Providers.Add<MyAuthorizationProvider>();
```

Authorization providers are registered to dependency injection automatically. An authorization provider can inject any dependency (like a repository) to build permission definitions using some other sources.

Checking Permissions

Using AbpAuthorize Attribute

The **AbpAuthorize** (**AbpMvcAuthorize** for MVC Controllers and **AbpApiAuthorize** for Web API Controllers) attribute is the easiest and most common way of checking permissions. Consider the application service method shown below:

Copy

```
[AbpAuthorize("Administration.UserManagement.CreateUser")]
public void CreateUser(CreateUserInput input)
{
    //A user can not execute this method if he is not granted the
"Administration.UserManagement.CreateUser" permission.
}
```

The CreateUser method can not be called by a user who is not granted the permission "*Administration.UserManagement.CreateUser*".

The AbpAuthorize attribute also checks if the current user is logged in (using IAbpSession.UserId). If we declare an AbpAuthorize for a method, it only checks for the login:

Copy

```
[AbpAuthorize]
public void SomeMethod(SomeMethodInput input)
{
    //A user can not execute this method if he did not login.
}
```

AbpAuthorize attribute notes

ASP.NET Boilerplate uses the power of dynamic method interception for authorization. There are some restrictions for the methods using the AbpAuthorize attribute.

- It cannot be used for private methods.
- It cannot be used for static methods.

- You can not use it for methods of a non-injected class (We must use dependency injection).

Also:

- You can use it for any **public** method if the method is called over an **interface** (like Application Services used over interface).
- A method should be **virtual** if it's called directly from a class reference (like ASP.NET MVC or Web API Controllers).
- A method should be **virtual** if it's **protected**.

Note: There are four types of authorize attributes:

- In an application service (application layer), we use the **Abp.Authorization.AbpAuthorize** attribute.
- In an MVC controller (web layer), we use the **Abp.Web.Mvc.Authorization.AbpMvcAuthorize** attribute.
- In ASP.NET Web API, we use the **Abp.WebApi.Authorization.AbpApiAuthorize** attribute.
- In ASP.NET Core, we use the **Abp.AspNetCore.Mvc.Authorization.AbpMvcAuthorize** attribute.

This difference comes from inheritance. In the application layer it's completely ASP.NET Boilerplate's implementation and it does not extend any class. For MVC and Web API, it inherits from the Authorize attributes of those frameworks.

Suppress Authorization

You can disable authorization for a method/class by adding **AbpAllowAnonymous** attribute to application services. Use the **AllowAnonymous** attribute for MVC, Web API and ASP.NET Core Controllers, which is a native attribute of these frameworks.

Using IPermissionChecker

While the AbpAuthorize attribute is good enough for most cases, there are situations where we may want to check for a permission in a method's body. We can inject and use **IPermissionChecker** for that as shown in the example below:

Copy

```
public void CreateUser(CreateOrUpdateUserInput input)
{
    if (!PermissionChecker.IsGranted("Administration.UserManagement.CreateUser"))
    {
        throw new AbpAuthorizationException("You are not authorized to create user!");
    }

    //A user can not reach this point if he is not granted for
"Administration.UserManagement.CreateUser" permission.
}
```

You can code any logic since **IsGranted** simply returns true or false (It has an Async version, too). If you simply check a permission and throw an exception as shown above, you can use the **Authorize** method:

Copy

```
public void CreateUser(CreateOrUpdateUserInput input)
{
    PermissionChecker.Authorize("Administration.UserManagement.CreateUser");

    //A user can not reach this point if he is not granted for
"Administration.UserManagement.CreateUser" permission.
}
```

Since authorization is widely used, **ApplicationService** and some common base classes inject and define the PermissionChecker property. Thus, permission checker can be used without injecting application service classes.

In Razor Views

The base view class defines the IsGranted method to check if the current user has permission. Thus, we can conditionally render the view. Example:

Copy

```
@if (IsGranted("Administration.UserManagement.CreateUser"))
{
    <button id="CreateNewUserButton" class="btn btn-primary"><i class="fa fa-plus"></i>
@L("CreateNewUser")</button>
}
```

Client Side (JavaScript)

In the client side, we can use the API defined in the **abp.auth** namespace. In most cases, we need to check if the current user has a specific permission (with permission name). Example:

Copy

```
abp.auth.isGranted('Administration.UserManagement.CreateUser');
```

You can also use **abp.auth.grantedPermissions** to get all granted permissions or **abp.auth.allPermissions** to get all available permission names in the application. Check **abp.auth** namespace on runtime for others.

Permission Manager

We may need the definitions of permissions. **IPermissionManager** can be injected and used in this case.