

Articles Tutorials

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

[Edit on GitHub](#)

- [User Entity](#)
- [User Manager](#)
 - [Multi-Tenancy](#)
 - [User Login](#)
 - [About IdentityResults](#)
- [External Authentication](#)
 - [LDAP/Active Directory](#)
 - [Social Logins](#)

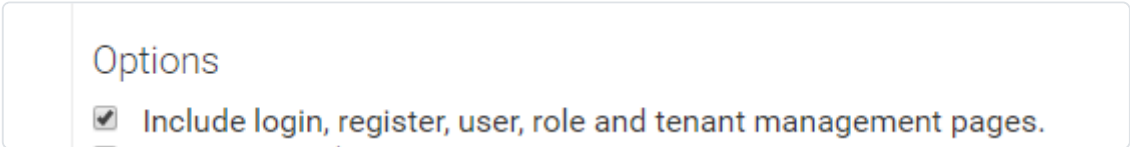
User Entity

The User entity represents a **user of the application**. It should be derived from the **AbpUser** class as shown below:

Copy

```
public class User : AbpUser<Tenant, User>
{
    //add your own user properties here
}
```

This class will be created when you download an ABP template with the option in the below image is selected.



Users are stored in the **AbpUsers** table in the database. You can add custom properties to the User class (and create database migrations for the changes).

The AbpUser class defines some base properties. Some of the properties are:

- UserName:** Login name of the user. Should be **unique** for a [tenant](#).
- EmailAddress:** Email address of the user. Should be **unique** for a [tenant](#).
- Password:** Hashed password of the user.
- IsActive:** True, if this user can login to the application.
- Name** and **Surname** of the user.

There are also some properties like **Roles**, **Permissions**, **Tenant**, **Settings**, **IsEmailConfirmed**, and so on. Check the AbpUser class for more information.

The AbpUser class is inherited from **FullAuditedEntity**. That means it has creation, modification and deletion audit properties. It's also implements [Soft-Delete](#) , so when we delete a user, it's not deleted from database, just marked as deleted.

The AbpUser class implements the [IMayHaveTenant](#) filter to properly work in a multi-tenant application.

Finally, the **Id** of the User is defined as **long**.

User Manager

userManager is a service to perform **domain logic** for users:

	Copy
<pre>public class userManager : AbpUserManager<Tenant, Role, User> { //... }</pre>	

You can [inject](#) and use userManager to create, delete, update users, grant permissions, change roles for users and much more. You can add your own methods here. Also, you can **override** any method of the **AbpUserManager** base class for your own needs.

Multi-Tenancy

If you're not creating a multi-tenant application, you can skip this section. See the [multi-tenancy documentation](#) for more information about multi-tenancy.

userManager is designed to work for a **single tenant** at a time. It works with the **current tenant** by default. Let's see some usages of the userManager:

	Copy
<pre>public class MyTestAppService : ApplicationService { private readonly userManager _userManager; public MyTestAppService(userManager userManager) { _userManager = userManager; } public void TestMethod_1() { //Find a user by email for current tenant var user = _userManager.FindByEmail("sampleuser@aspnetboilerplate.com"); } public void TestMethod_2() { //Switch to tenant 42 CurrentUnitOfWork.SetFilterParameter(AbpDataFilters.MayHaveTenant, AbpDataFilters.Parameters.TenantId, 42); //Find a user by email for tenant 42 var user = _userManager.FindByEmail("sampleuser@aspnetboilerplate.com"); } public void TestMethod_3() { //Disabling MayHaveTenant filter, so we can reach all users using (CurrentUnitOfWork.DisableFilter(AbpDataFilters.MayHaveTenant)) { //Now, we can search for a username in all tenants var users = _userManager.Users.Where(u => u.UserName == "sampleuser").ToList(); //Or we can add TenantId filter if we want to search for a specific tenant var user = _userManager.Users.FirstOrDefault(u => u.TenantId == 42 && u.UserName == "sampleuser"); } } }</pre>	

User Login

Module Zero defines LoginManager which has a **LoginAsync** method used for logging into the application. It checks all logic for the login and returns a login result. The LoginAsync method also **automatically saves all login attempts** to the database (even if it's a failed attempt). You can use the **UserLoginAttempt** entity to query it.

About IdentityResults

Some methods of UserManager return IdentityResult as a result instead of throwing exceptions for some cases. This is the nature of ASP.NET Identity Framework. Module Zero also follows it, so we should check this returning result object to know if the operation succeeded.

Module Zero defines the **CheckErrors** extension method that automatically checks errors and throws an exception (a localized [UserFriendlyException](#)) if needed. Example usage:

	Copy
<pre>(await UserManager.CreateAsync(user)).CheckErrors();</pre>	

To get localized exceptions, we must provide a [ILocalizationManager](#) instance:

	Copy
<pre>(await UserManager.CreateAsync(user)).CheckErrors(LocalizationManager);</pre>	

External Authentication

The Login method of Module Zero authenticates a user from the **AbpUsers** table in the database. Some applications may require you to authenticate users from some external sources (like active directory, from another database's tables, or even from a remote service).

For such cases, UserManager defines an extension point named 'external authentication source'. We can create a class derived from **IExternalAuthenticationSource** and register it to the configuration. There is a **DefaultExternalAuthenticationSource** class to simplify the implementation of IExternalAuthenticationSource. Let's see an example:

	Copy
<pre>public class MyExternalAuthSource : DefaultExternalAuthenticationSource<Tenant, User>, ITransientDependency { public override string Name { get { return "MyCustomSource"; } } public override Task<bool> TryAuthenticateAsync(string userNameOrEmailAddress, string plainPassword, Tenant tenant) { //TODO: authenticate user and return true or false } }</pre>	

In the TryAuthenticateAsync method, we can check the username and password from some source and return true if a given user is authenticated by it. We can also override the CreateUser and UpdateUser methods to control user creation and updating for this source.

When a user is authenticated by an external source, Module Zero checks if this user exists in the database (AbpUsers table). If not, it calls CreateUser to create the user, otherwise it calls UpdateUser to allow the authentication source to update existing user information.

We can define more than one external authentication source in an application. The AbpUser entity has an AuthenticationSource property that shows which source authenticated this user.

To register our authentication source, we can use some code like this in the [PreInitialize](#) method of our module:

	Copy
<pre>Configuration.Modules.Zero().UserManagement.ExternalAuthenticationSources.Add<MyExternalAuthS ();</pre>	

LDAP/Active Directory

LdapAuthenticationSource is an implementation of external authentication to make users login with their LDAP (active directory) username and password.

If we want to use LDAP authentication, we must first add the [Abp.Zero.Ldap](#) NuGet package to our project (generally to the Core (domain) project). We then must extend the **LdapAuthenticationSource** for our application as shown below:

	Copy
<pre>public class MyLdapAuthenticationSource : LdapAuthenticationSource<Tenant, User> { public MyLdapAuthenticationSource(ILdapSettings settings, IAbpZeroLdapModuleConfig ldapModuleConfig) : base(settings, ldapModuleConfig) { } }</pre>	

Lastly, we must set a module dependency to **AbpZeroLdapModule** and **enable** LDAP with the auth source created above:

	Copy
<pre>[DependsOn(typeof(AbpZeroLdapModule))] public class MyApplicationCoreModule : AbpModule { public override void PreInitialize() { Configuration.Modules.ZeroLdap().Enable(typeof (MyLdapAuthenticationSource)); } ... }</pre>	

After these steps, the LDAP module will be enabled for your application, but LDAP auth is not enabled by default. We can enable it using the settings.

By default, LDAP authentication uses **SamAccountName** for the username. If you want to use **UserPrincipalName**, you can configure **Configuration.Modules.ZeroLdap().UseUserPrincipalNameAsUserName** to **true**.

Settings

The **LdapSettingNames** class defines constants for setting names. You can use these constant names while changing settings (or getting settings). LDAP settings are **per-tenant** (for multi-tenant applications), so different tenants have different settings (see the setting definitions on [github](#)).

As you can see in the MyLdapAuthenticationSource **constructor**, LdapAuthenticationSource expects **ILdapSettings** as a constructor argument. This interface is used to get the LDAP settings like domain, user name and password to connect to Active Directory. The default implementation (**LdapSettings** class) gets these settings from the [setting manager](#).

If you work with Setting manager, then there's no problem. You can change the LDAP settings using the [setting manager API](#). If you want, you can add some initial seed data to the database to enable LDAP auth by default.

Note: If you don't define a domain, username and password, LDAP authentication works for the current domain if your application runs in a domain with appropriate privileges.

Custom Settings

If you want to define another setting source, you can implement a custom ILdapSettings class as shown below:

	Copy
<pre>public class MyLdapSettings : ILdapSettings { public async Task<bool> GetIsEnabled(int? tenantId) { return true; } public async Task<ContextType> GetContextType(int? tenantId) { return ContextType.Domain; } public async Task<string> GetContainer(int? tenantId) { return null; } public async Task<string> GetDomain(int? tenantId) { return null; } public async Task<string> GetUserName(int? tenantId) { return null; } public async Task<string> GetPassword(int? tenantId) { return null; } }</pre>	

Then register it to IOC in PreInitialize method of your module:

	Copy
<pre>[DependsOn(typeof(AbpZeroLdapModule))] public class MyApplicationCoreModule : AbpModule { public override void PreInitialize() { IocManager.Register<ILdapSettings, MyLdapSettings>(); //change default setting source Configuration.Modules.ZeroLdap().Enable(typeof (MyLdapAuthenticationSource)); } ... }</pre>	

Then you can get the LDAP settings from another source.

Social Logins


See the [social authentication](#) document for more info about social logins.



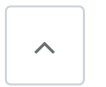
 Stars  10k

Follow

3,750 followers

 [Stack Overflow](#)

2013 - 2022 © [Volosoft](#)



Copyright © 2021 .NET Foundation