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

Edit on GitHub

In this document

- Introduction
  - About ISettingStore
- Defining settings
  - Setting scope
  - Overriding Setting Definitions
- Getting setting values
  - Server-side
  - Client-side
- Changing settings
- ISettingEncryptionService
- About caching

Introduction

Every application needs to store some settings and use these settings somewhere in the application. ASP.NET Boilerplate provides a strong infrastructure to store/retrieve **application**, **tenant** and **user** level settings available on both the **server** and **client** sides.

A setting is a **name-value string** pair that is generally stored in a database (or another source). We can store non-string values by converting it to a string.

About ISettingStore

The **ISettingStore** interface must be implemented in order to use the setting system. While you can implement it in your own way, it's fully implemented in the **Module Zero** project. If it's not implemented, settings are read from the application's **configuration file** (web.config or app.config) but those settings cannot be changed. Scoping will also not work.

Defining settings

A setting must be defined before its use. ASP.NET Boilerplate is designed to be modular, so different modules can have different settings. A module must create a class derived from the **SettingProvider** in order to define its settings. An example setting provider is shown below:

Copy

Authorization
Feature Management
Audit Logging
Entity History
Distributed Service Layer
ASP.NET Web API
Web API Controllers
Dynamic Web API Layer
OData Integration

```
public class MySettingProvider : SettingProvider
{
    public override IEnumerable<SettingDefinition>
GetSettingDefinitions(SettingDefinitionProviderContext context)
    {
        return new[]
                {
                    new SettingDefinition(
                        "SmtpServerAddress",
                        "127.0.0.1"
                        ),

                    new SettingDefinition(
                        "PassiveUsersCanNotLogin",
                        "true",
                        scopes: SettingScopes.Application | SettingScopes.Tenant
                        ),

                    new SettingDefinition(
                        "SiteColorPreference",
                        "red",
                        scopes: SettingScopes.User,
                        clientVisibilityProvider: new
VisibleSettingClientVisibilityProvider()
                        )

                };
    }
}
```

The **GetSettingDefinitions** method returns the **SettingDefinition** objects. The SettingDefinition class has some parameters in it's constructor:

- **Name** (required): A setting must have a system-wide **unique** name. It's a good idea to define a const string for a setting name instead of using a magic string.
- **Default value**: A setting may have a default value. This value can be null or an empty string.
- **Scopes**: A setting should define it's scope (see below).
- **Display name**: A localizable string that can be used to show setting's name later in the UI.
- **Description**: A localizable string that can be used to show a setting's description later in the UI.
- **Group**: Can be used to group settings. This is just for the UI, and not used in setting management.
- **ClientVisibilityProvider**: Can be used to determine if a setting can be used on the client-side or not.
- **isInherited**: Used to set if this setting is inherited by tenant and users (See setting scope section).
- **customData**: Can be used to set custom data for this setting definition.
- **IsEncrypted**: A Boolean value indicates that whether this setting value should be encrypted on save and decrypted on read. It makes possible to secure the setting value in the database.

After creating a setting provider, we must register it in the PreIntialize method of our module:

Copy

```
Configuration.Settings.Providers.Add<MySettingProvider>();
```

The setting providers are registered via dependency injection automatically. A setting provider can inject any dependency (like a repository) to build the setting definitions using some other source.

Setting encryption/decryption are done using `SettingEncryptionConfiguration`. In order to change default encryption/decryption configuration, you can access this configuration as below;

Copy

```
Configuration.Settings.SettingEncryptionConfiguration.Keysize = 256;
Configuration.Settings.SettingEncryptionConfiguration.DefaultPassPhrase = "pass_phrase";
Configuration.Settings.SettingEncryptionConfiguration.InitVectorBytes = Encoding.ASCII.GetByt
Configuration.Settings.SettingEncryptionConfiguration.DefaultSalt = Encoding.ASCII.GetBytes('
```

Setting scope

There are three **setting scopes** (or levels) defined in the **SettingScopes** enum:

- **Application**: An application scoped setting is used for user/tenant independent settings. For example, we can define a setting named "SmtpServerAddress" to get the server's IP address when sending emails. If this setting has a single value (not changes based on users), then we can define it as Application scoped.
- **Tenant**: If the application is multi-tenant, we can define tenant-specific settings.
- **User**: We can use a user-scoped setting to store/get the value of the setting specific to each user.

The SettingScopes enum has a **Flags** attribute, so we can define a setting with **more than one scope**.

The setting scope is **hierarchic** by default (unless you set **isInherited** to false). For example, if we define a setting's scope as "Application | Tenant | User" and try to get **current value** of the the setting;

- We get the user-specific value if it's defined (overrided) for the user.
- If not, we get the tenant-specific value if it's defined (overrided) for the tenant.
- If not, we get the application value if it's defined.
- If not, we get the **default value**.

The default value can be **null** or an **empty** string. It's recommended that you provide default values for settings where it's possible.

Overriding Setting Definitions

context.Manager can be used to get a setting definition to change its values. In this way, you can manipulate setting definitions of dependent modules.

Getting setting values

After defining a setting, we can get its current value on both the server and client.

Server-side

ISettingManager

The **ISettingManager** is used to perform setting operations. We can inject and use it anywhere in the application. ISettingManager defines many methods to get a setting's value.

The most-used method is **GetSettingValue** (or GetSettingValueAsync for an async call). It returns the **current value** of the setting based on the default value, application, tenant and user settings (as described in Setting scope section before). Examples:

Copy

```
//Getting a boolean value (async call)
var value1 = await SettingManager.GetSettingValueAsync<bool>("PassiveUsersCanNotLogin");

//Getting a string value (sync call)
var value2 = SettingManager.GetSettingValue("SmtpServerAddress");
```

GetSettingValue has generic and async versions as shown above. There are also methods to get a specific tenant or user's setting value or list of all setting values.

Since ISettingManager is widely used, some special **base classes** (like ApplicationService, DomainService and AbpController) have a property named **SettingManager**. If we derive from these classes, there's no need to explicitly inject it.

ISettingDefinitionManager

Also `ISettingDefinitionManager` can be used to get setting definitions that are defined in `AppSettingProvider`. We can inject and use it anywhere in the application as well. You can get definition name, default value, display name and etc. by using `ISettingDefinitionManager`.

Client-side

ClientVisibilityProvider property of a setting definition determines the visibility of a setting for the client-side. There are four implementations of ISettingClientVisibilityProvider.

- **VisibleSettingClientVisibilityProvider**: Makes a setting definition visible to the client-side.
- **HiddenSettingClientVisibilityProvider**: Makes a setting definition hidden to the client-side.
- **RequiresAuthenticationSettingClientVisibilityProvider**: Makes a setting definition visible to the client-side if a user is logged in.
- **RequiresPermissionSettingClientVisibilityProvider**: Makes a setting definition visible to the client side if logged in user has a specific permission.

If a setting is visible to client-side according to ClientVisibilityProvider of a setting definition , then you can get it's current value on the client-side using JavaScript. The **abp.setting** namespace defines the needed functions and objects. Example:

Copy

```
var currentColor = abp.setting.get("SiteColorPreference");
```

There are also the **getInt** and **getBoolean** methods. You can get all the values using the **abp.setting.values** object. Note that if you change a setting on the server-side, the client can not know this change unless the page is refreshed, settings are somehow reloaded, or manually updated by code.

Changing settings

The ISettingManager defines the **ChangeSettingForApplicationAsync**, **ChangeSettingForTenantAsync** and **ChangeSettingForUserAsync** methods (and sync versions) to change settings for the application, for a tenant and for a user respectively.

ISettingEncryptionService

`ISettingEncryptionService` is used to encrypt/decrypt setting values when `IsEncrypted` property of a setting definition was set to `true`.

You can replace this service in the dependency injection system to customize the encryption/decryption process.

About caching