2.5. Setting Management

2.5.1. Introduction

Every application need to store some settings and use these settings in somewhere in the application. StudioX provides a strong infrastructure to store/retrieve application, tenant and user level settings usable both in 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 to string.

About ISettingStore

ISettingStore interface must be implemented in order to use setting system. While you can implement it in your own way, it's fully implemented in module-zero project. If it's not implemented, settings are read from application's configuration file (web.config or app.config) but can not change any setting. Also, scoping will not work.

2.5.2. Defining settings

A setting must be defined before usage. StudioX is designed to be modular. So, different modules can have different settings. A module should create a class derived from SettingProvider in order to define it's settings. An example setting provider is shown below:

```
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, isVisibleToClients: true)
                };
    }
}
```

GetSettingDefinitions method should return SettingDefinition objects. SettingDefinition class has some parameters in it's constructor:

- **Name** (required): A setting must have a system-wide unique name. It's good idea to define a const string for a setting name instead of a magic string.
- **Default value**: A setting may have a default value. This value can be null or 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 UI.
- **Description**: A localizable string that can be used to show setting's description later in UI.
- **Group**: Can be used to group settings. This is just for UI, not used in setting management.
- **IsVisibleToClients**: Set true to make a setting usable on the client side.
- **isInherited**: Used to set if this setting is inherited by tenant and users (See setting scope section).
- **customData**: Can be used to set a custom data for this setting definition.

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

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

Setting providers are registered to dependency injection automatically. So, a setting provider can inject any dependency (like a repository) to build setting definitions using some other sources.

Setting scope

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

- **Application**: An application scoped setting is used for user/tenant independed settings. For example, we can define a setting named "SmtpServerAddress" to get 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 value of the setting specific to each user.

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

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.

Default value can be null or empty string. It's adviced to provide default values for settings where it's possible.

Overriding Setting Definitions

context.Manager can be used to get a setting definition to change it's values. In this way, you can manipulate setting definitions of depended modules.

Getting setting values

After defining a setting, we can get it's current value both in server and client.

Server side

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.

Most used method is GetSettingValue/GetSettingValueAsync. It returns current value of the setting based on default value, application, tenant and user settings (as described in Setting scope section before). Examples:

```
//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 StudioXController) has a property named SettingManager. If we derived from these classes, no need to explicitly inject it.

Client side

If you set IsVisibleToClients as true while defining a setting, then you can get it's current value in the client side using javascript. StudioX.setting namespace defines needed functions and objects. Example:

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

There is also getInt and getBoolean methods. You can get all values using StudioX.setting.values object. Note that; If you change a setting in server side, clienct can not know this change unless page is refreshed, settings are somehow reloaded or it's manually updated by code.

2.5.3. Changing settings

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

2.5.4. About caching

Setting Manager caches settings on the server side. So, we should not directly change a setting value using repository or database update query.