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

Introduction

ASP.NET Boilerplate defines a strong UI localization system which is used both on the server and client sides. It allows us to easily configure application languages and define localization texts (strings) in different sources (Resource files and XML files are two pre-defined sources).

While it's good for most cases, we may want to define languages and texts **dynamically** and on a **database**. Module Zero allows us to dynamically manage application **languages** and **texts per tenant**.

About Localization

We strongly recommend you read the localization documentation before this document.

How To Enable

Startup Template

If you create your project from the startup templates, you can skip this section since the template comes with the database-based localization enabled by default. If you created your project before this feature, please read this to enable it for your application.

Database localization is designed to be backwards-compatible with ASP.NET Boilerplate's existing localization system. It actually replaces all the existing dictionary-based localization sources with **MultiTenantLocalizationSource**.

MultiTenantLocalizationSource wraps existing **DictionaryBasedLocalizationSource** based sources. We generally wrap XML based localization sources. It can not wrap **Resource File** sources since resource files are designed as hard-coded and static files which are not proper for dynamic localization.

Since it's a wrapper, the underlying XML files are used as a fallback source if a text is not localized in the database. It may seem complicated, but it's easy to implement for your application. Let's see how to enable the database-based localization.

EnableDbLocalization

First, we enable it:

```
Configuration.Modules.Zero().LanguageManagement.EnableDbLocalization();
```

This should be done in the **top level** module's **PreInitialize** method (it's the web module for a web application. Import the Abp.Zero.Configuration namespace (using Abp.Zero.Configuration) to see the Zero() extension methods).

This configuration makes all the magic happen, but we must add some more code to make it work properly.

Seed Database Languages

Since ABP will get a list of languages from the database, we must **insert** the default languages into it. If you're using EntityFramework, you can use this seed code:

Remove Static Language Configuration

If you have a static language configuration like the one shown below, you can **delete** these lines from your configuration code, since they will get the languages from the database.

```
Configuration.Localization.Languages.Add(new LanguageInfo("en", "English", "famfamfam-flag-england", true));
```

Note On Existing XML Localization Sources

Do not delete your XML localization files and source configuration code. These files are used as a **fallback source** and all localization keys are obtained from this source.

So when you need a new localized text, **define it** into the XML files as you do normally. You must at least define it in the **default** language's XML file. Note: you don't need to add the default values of the localized texts to the database migration code.

Managing Languages

The **IApplicationLanguageManager** interface is injected and used to manage languages. It has methods like GetLanguagesAsync, AddAsync, RemoveAsync, UpdateAsync... to manage languages for the host and tenants.

Language List Logic

The list of languages are stored per tenant and for the host, and calculated as follows:

- There is a list of languages defined for **the host**. This list is considered as the **default** for all tenants.
- There is a separated list of languages for **each tenant**. This list **inherits** the host list and **adds** tenant-specific languages. Tenants can not delete or update host-defined (default) languages (but can override localization texts as we will see later).

ApplicationLanguage Entity

The ApplicationLanguage entity represents a language for a tenant or the host.

```
[Serializable]
[Table("AbpLanguages")]
public class ApplicationLanguage : FullAuditedEntity, IMayHaveTenant
{
    //...
}
```

Its basic properties are:

- **TenantId** (nullable): Contains the related tenant's Id if this language is tenant-specific. It's null if this is a host language.
- **Name**: Name of the language. This **must be a culture code** from this list.
- **DisplayName**: Shown name of the language. This can be an arbitrary name, but it generally is the CultureInfo.DisplayName.
- **Icon**: An arbitrary icon/flag for the language. This can be used to show flag of the language on the UI.

The ApplicationLanguage also inherits from **FullAuditedEntity**. This means it's a **soft-delete** entity and automatically **audited** (see the entity document for more info).

The ApplicationLanguage entities are stored in the **AbpLanguages** table in the database.

Managing Localization Texts

The **IApplicationLanguageTextManager** interface is injected and used to manage localization texts. It has the needed methods to get/set a localization text for a tenant or the host.

Localizing A Text

Let's see what happens when you want to localize a text;

- First, it tries to get the **current culture** using the CurrentThread.CurrentUICulture.
  - It checks if the given text is defined (overridden) for the **current tenant** using IAbpSession.TenantId in the **current culture** in the database. It returns the value if it is defined.
  - It then checks if a given text is defined (overridden) for the **host** in the **current culture** in the database. It returns the value if it is defined.
  - It then checks if a given text is defined in the underlying XML file in the **current culture**. It returns the value if it is defined.
- Second, it will try to find the **fallback culture**. It's calculated like this: If the current culture is "en-GB", then the fallback culture is "en".
  - It checks if a given text is defined (overrided) for the **current tenant** in the **fallback culture** in the database. It returns the value if it is defined.
  - It then checks if the given text is defined (overridden) for the **host** in the **fallback culture** in the database. It returns the value if it is defined.
  - It then checks if the given text is defined in the underlying XML file in the **fallback culture**. It returns the value if it is defined.
- Third, it will try to find the **default culture**.
  - It checks if a given text is defined (overridden) for the **current tenant** in the **default culture** in the database. It returns the value if it is defined.
  - It then checks if the given text is defined (overridden) for the **host** in the **default culture** in the database. It returns the value if it is defined.
  - It then checks if a given text is defined in the underlying XML file in the **default culture**. It returns the value if it is defined.
- If all attempts fail, it will get the same text or an throw exception.

- If the given text (key) is not found at all, ABP throws an exception or returns the same text (key) by wrapping it with [and] (it can be configured on startup, see the localization document).

Getting a localized text is a bit complicated, but it works fast since it uses the cache.

ApplicationLanguageText Entity

The ApplicationLanguageText entity is used to store localized values in the database.

Copy

```
[Serializable]
[Table("AbpLanguageTexts")]
public class ApplicationLanguageText : AuditedEntity<long>, IMayHaveTenant
{
    //...
}
```

It's basic properties are;

- **TenantId** (nullable): Contains the related tenant's Id if this localized text is tenant-specific. It's null if this is a host-localized text.
- **LanguageName**: Name of the language. This **must be a culture code** from this list. This matches to the ApplicationLanguage.Name but is not a forced foreign key to make it independent from the language entry. IApplicationLanguageTextManager handles it properly.
- **Source**: Localization source name.
- **Key**: Localization text's key/name.
- **Value**: Localized value.

ApplicationLanguageText entities are stored in the **AbpLanguageTexts** table in the database.