SqlLocalizer

Release 1.0

SqlLocalizer

May 14, 2017

Contents

CHAPTER 1

SQL Localization for ASP.NET Core, supports all EF Core providers.

Pull requests welcome!

Overview

Simple SQL Localization NuGet package which can be used with ASP.NET Core and any database supported by Entity Framework Core. The localization can be used like the default ASP.NET Core localization.

Features

- Supports any Entity Framework Core (EF Core) persistence
- Import, export
- Cache, reset cache
- support for live update
- Configurable keys for localization records
- Default key display possible, if no localization available
- KISS

NuGet package

https://www.nuget.org/packages/Localization.SqlLocalizer/

Source code

https://github.com/damienbod/AspNetCoreLocalization/tree/master/src/Localization.SqlLocalizer

Issues

https://github.com/damienbod/AspNetCoreLocalization/issues

Quickstart

Download the package from NuGet and add it to your project.json file.

```
"dependencies": {
  "Localization.SqlLocalizer": "1.0.6",
        ...
```

Select your Entity Framework Core (EF Core) provider and also add to the project.json file:

```
"dependencies": {
  "Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore": "1.1.0",
  "Microsoft.EntityFrameworkCore.Sqlite": "1.1.0",
  "Microsoft.EntityFrameworkCore.Sqlite.Design": "1.1.0",
        ...

"tools": {
  "Microsoft.EntityFrameworkCore.Tools.DotNet": "1.1.0-preview4",
        ...
},
```

See EF Core for more details on installing updating a provider

https://blogs.msdn.microsoft.com/dotnet/2016/11/16/announcing-entity-framework-core-1-1/

Add the configuration to the Startup:

```
public void ConfigureServices(IServiceCollection services)
{
        // init database for localization
        var sqlConnectionString = Configuration["DbStringLocalizer:ConnectionString"];

        services.AddDbContext<LocalizationModelContext>(options =>
                options.UseSqlite(
                        sqlConnectionString,
                        b => b.MigrationsAssembly("AspNet5Localization")
                )
        );

        var useTypeFullNames = true;
        var useOnlyPropertyNames = false;
        var returnOnlyKeyIfNotFound = false;

        // Requires that LocalizationModelContext is defined
        services.AddSqlLocalization(options => options.UseSettings(useTypeFullNames,␣
↪useOnlyPropertyNames, returnOnlyKeyIfNotFound));
        // services.AddSqlLocalization(options => options.ReturnOnlyKeyIfNotFound =␣
↪true);
        // services.AddLocalization(options => options.ResourcesPath = "Resources");

        services.AddMvc()
                .AddViewLocalization()
                .AddDataAnnotationsLocalization();
```

```
        services.AddScoped<LanguageActionFilter>();

        services.Configure<RequestLocalizationOptions>(
                options =>
                        {
                                var supportedCultures = new List<CultureInfo>
                                {
                                        new CultureInfo("en-US"),
                                        new CultureInfo("de-CH"),
                                        new CultureInfo("fr-CH"),
                                        new CultureInfo("it-CH")
                                };

                                options.DefaultRequestCulture = new␣
↪RequestCulture(culture: "en-US", uiCulture: "en-US");
                                options.SupportedCultures = supportedCultures;
                                options.SupportedUICultures = supportedCultures;
                        });
}
```

And also the Configure method in the Startup.cs:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env,␣
↪ILoggerFactory loggerFactory)
{
        loggerFactory.AddConsole();
        loggerFactory.AddDebug();

        var locOptions = app.ApplicationServices.GetService<IOptions
↪<RequestLocalizationOptions>>();
        app.UseRequestLocalization(locOptions.Value);

        app.UseStaticFiles();

        app.UseMvc();
}
```

Use migrations to create the database if required:

```
dotnet ef migrations add Localization --context LocalizationModelContext

dotnet ef database update Localization --context LocalizationModelContext
```

Use like the standard localization:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Localization;

namespace AspNet5Localization.Controllers
{
        [Route("api/[controller]")]
        public class AboutController : Controller
        {
                private readonly IStringLocalizer<SharedResource> _localizer;
                private readonly IStringLocalizer<AboutController> _
↪aboutLocalizerizer;
```

```
                public AboutController(IStringLocalizer<SharedResource> localizer,␣
↪IStringLocalizer<AboutController> aboutLocalizerizer)
                {
                        _localizer = localizer;
                        _aboutLocalizerizer = aboutLocalizerizer;
                }

                [HttpGet]
                public string Get()
                {
                        // _localizer["Name"]
                        return _aboutLocalizerizer["AboutTitle"];
                }
        }
}
```

Configuration

The following configurations can be set in the ConfigureServices method in the Startup.cs file.

The LocalizationModelContext needs to be added to the services. You need to then decide which database is to be used. Because the context is in a separate assembly, if you are using migrations, you need to define the MigrationsAssembly property.

ConfigureServices in the Startup.cs:

```
public void ConfigureServices(IServiceCollection services)
{
        // init database for localization
        var sqlConnectionString = Configuration["DbStringLocalizer:ConnectionString"];

        services.AddDbContext<LocalizationModelContext>(options =>
                options.UseSqlite(
                        sqlConnectionString,
                        b => b.MigrationsAssembly("Angular2LocalizationAspNetCore")
                )
        );

        // Requires that LocalizationModelContext is defined
        services.AddSqlLocalization(options => options.UseTypeFullNames = true);
```

Default

Uses the Name property of the class type to create the key:

```
services.AddSqlLocalization();
```

Using full types as keys

Uses the FullName of the class type to create the key:

```
services.AddSqlLocalization(options => options.UseTypeFullNames = true);
```

Using only property names

Uses only the property name for the key:

```
services.AddSqlLocalization(options => options.UseOnlyPropertyNames = true);
```

Displaying default keys

Display default keys when localization is undefined:

```
var useTypeFullNames = true;
var useOnlyPropertyNames = false;
var returnOnlyKeyIfNotFound = true;

services.AddSqlLocalization(options => options.UseSettings(
  useTypeFullNames,
  useOnlyPropertyNames,
  returnOnlyKeyIfNotFound,
  false
));
```

Development add resources automatically if not found

You should only use this feature in development. The env.IsDevelopment() method provides a good way to configure this.

Add when undefined:

```
private bool _createNewRecordWhenLocalisedStringDoesNotExist = false;

public Startup(IHostingEnvironment env)
{
        var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional:␣
↪true);

        builder.AddEnvironmentVariables();
        Configuration = builder.Build();

        if (env.IsDevelopment())
        {
                _createNewRecordWhenLocalisedStringDoesNotExist = true;
        }
}

public void ConfigureServices(IServiceCollection services)
{
        var useTypeFullNames = false;
        var useOnlyPropertyNames = false;
```

```
        var returnOnlyKeyIfNotFound = false;

        services.AddSqlLocalization(options => options.UseSettings(
                useTypeFullNames,
                useOnlyPropertyNames,
                returnOnlyKeyIfNotFound,
                _createNewRecordWhenLocalisedStringDoesNotExist));
```

Setting the schema

Set the SQL setting for the localization:

```
services.AddLocalizationSqlSchema("translations");
services.AddDbContext<LocalizationModelContext>(options =>
        options.UseSqlite(
                sqlConnectionString,
                b => b.MigrationsAssembly("AspNet5Localization")
        )
);
```

Importing and Exporting

IStringExtendedLocalizerFactory Import, Export methods

The UpdatetLocalizationData and the AddNewLocalizationData methods of the IStringExtendedLocalizerFactory can be used to import and export data into the database. If these methods are used, the cache is reset. As a user of this interface, you have to implement the logic to decide whether a localization record needs to be updated or added.

```
public interface IStringExtendedLocalizerFactory : IStringLocalizerFactory
{
    IList GetImportHistory();

    IList GetExportHistory();

    IList GetLocalizationData(string reason = "export");

    IList GetLocalizationData(DateTime from, string culture = null, string reason =
↪"export");

    void UpdatetLocalizationData(List<LocalizationRecord> data, string information);

    void AddNewLocalizationData(List<LocalizationRecord> data, string information);
```

The following example shows how the import, export could be implemented using csv.

https://damienbod.com/2016/07/15/import-export-asp-net-core-localized-data-as-csv/

Cache Reset, live updates

If you decide to update the localization data directly in the database, you can reset the application localization cache using the ResetCache methods from the IStringExtendedLocalizerFactory interface.

By using these methods, the application does not need to be restarted to update the localization values.

IStringExtendedLocalizerFactory ResetCache methods

```
public interface IStringExtendedLocalizerFactory : IStringLocalizerFactory
{
    void ResetCache();

    void ResetCache(Type resourceSource);
```

IStringExtendedLocalizerFactory

When using the Localization.SqlLocalizer package, the IStringExtendedLocalizerFactory can be used for extra features of this package which are not included in the IStringLocalizerFactory.

```
public interface IStringExtendedLocalizerFactory : IStringLocalizerFactory
{
    void ResetCache();

    void ResetCache(Type resourceSource);

    IList GetImportHistory();

    IList GetExportHistory();

    IList GetLocalizationData(string reason = "export");

    IList GetLocalizationData(DateTime from, string culture = null, string reason =
↪"export");

    void UpdatetLocalizationData(List<LocalizationRecord> data, string information);

    void AddNewLocalizationData(List<LocalizationRecord> data, string information);

}
```

Example using the interface:

```
[Route("api/ImportExport")]
public class ImportExportController : Controller
{
    private IStringExtendedLocalizerFactory _stringExtendedLocalizerFactory;

    public ImportExportController(IStringExtendedLocalizerFactory␣
↪stringExtendedLocalizerFactory)
    {
        _stringExtendedLocalizerFactory = stringExtendedLocalizerFactory;
    }

    [HttpGet]
    [Route("localizedData.csv")]
    [Produces("text/csv")]
    public IActionResult GetDataAsCsv()
    {
```

```
        return Ok(_stringExtendedLocalizerFactory.GetLocalizationData());
    }
```

Links Docs, Examples

ASP.NET Core Localization example

https://github.com/damienbod/AspNetCoreLocalization/tree/master/src/AspNetCoreLocalization

Angular Localization with ASP.NET Core

https://github.com/damienbod/Angular2LocalizationAspNetCore

This article shows how localized data can be imported and exported using Localization.SqlLocalizer. The data is exported as CSV using the Formatter defined in the WebApiContrib.Core.Formatter.Csv package. The data can be imported using a file upload.

https://damienbod.com/2016/07/15/import-export-asp-net-core-localized-data-as-csv/

Offical docs from Microsoft: Globalization and localization

https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization

EF Core Providers

https://docs.microsoft.com/en-us/ef/core/providers/

Release History

Note: Current Version using ASP.NET Core 1.1, EFCore 1.1

Version 1.0.10

- Automatically add undefined resources for development
- Support for net461

Version 1.0.9

- Updating to VS2017 and csproj, .NET 1.1.1

Version 1.0.7

- Support for SQL schemas

Version 1.0.6

- return default key if localization cannot be found support

Example:

```
var useTypeFullNames = true;
var useOnlyPropertyNames = false;
var returnOnlyKeyIfNotFound = true;

services.AddSqlLocalization(options => options.UseSettings(
  useTypeFullNames,
  useOnlyPropertyNames,
  returnOnlyKeyIfNotFound
));
```

Version 1.0.5

- bugfix context System.InvalidOperationException import, export

Version 1.0.4

- Updated to .NET Core 1.1
- changed the constraint to included the resourceKey for new records

Version 1.0.3

- adding import, export interfaces

Version 1.0.2

- Updated to dotnet RTM

Version 1.0.1

- Added Unique constraint for key, culture
- Fixed type full name cache bug

Version 1.0.0

- Initial release
- Runtime localization updates
- Cache support, reset cache
- ASP.NET DI support
- Supports any Entity Framework Core database