ProDinner – ASP.net Core Awesome demo application

We presume that you will have the ProDinnerCore solution open while reading this so that you could read and look at the source.

Project structure:

- Core – entities
- Data – data access layer, contains the configuration for EF Core (DbContext)
- Service – contains services that use the DbContext, and some infra utils like password hashing and image editing.
- WebUI – web user interface

Data Access

For data access Entity Framework Core is being used. All the entity properties are named the same as the column names in the DB. Also the relationships are made accordingly to the EF default conventions. Table names are determined based on the DbSet<T> properties names on the ProDbContext. Because we have many to many relationships for dinner <-> meal and user <-> role and this isn't fully supported in EFCore we have some custom configuration inside ProDbContext.`OnModelCreating`.

```
The connection string is in appsettings.json and in Startup.cs we have a call to:
services.AddDbContext<ProDbContext>
```

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

Service Layer

Here we do the CRUD operations using the ProDbContext from Data. We also have some logic for caching, hashing passwords and resizing and saving images.

Cache

We have a CacheManager defined in our Service layer, inside it the entities dependencies are hardcoded so for example if there's a change for the Meals besides emptying the cache for Meals we also empty the cache for Dinners because a Dinner has many Meals.

All the Create and Edit (Delete is also Edit, sets IsDeleted = true) are done in CrudService<T> and inside each method that creates or edits an entity we call cache.ChangeAction which will reset the cache, in the case of Create it won't reset the cache for dependent entities (if we create a new meal, we don't empty the cache for Dinners).

Multilanguage User Interface

It is done using resource files. We're using a single resource Mui for all our localization. The resx files are located in WebUI/Resources, a path that is specified in Startup.cs:

```
services.AddLocalization(opts => { opts.ResourcesPath = "Resources"; });
```

In order to use one single resource file for all our views we have a `SharedViewLocalizer` which you can see added in ViewImports and Startup.cs.

In the controller we use `IStringLocalizer<Mui> (see DataController.cs for example), and for DataAnnotations see Startup.cs call to AddDataAnnotationsLocalization.`

For use in js we generate a dictionary in ClientUtil.GetClientDict and set this dictionary on the site (site.js) client object in _Layout.cshtml

ClientUtils GetCulture HtmlHelper extension is used to reference the correct awedict.js in _Layout.cshtml.

And finally in AwesomeConfig.cs we set `Settings.GetText, this will do all the server side localization for the awesome html helpers. AwesomeConfig is called in Startup.cs.`

Background Worker

There is `RestoreTimedHostedService`, which you can see registered in Startup, it periodically restores deleted entities based on some basic rules. It is a singleton but it needs CrudService which is a scoped instance that's why we use ServiceProvider to create a scope, and use that scope to resolve the CrudService on each work action.

WebUI

Has the controllers, views, scripts, content (css, images) and some stuff in Global.asax for app initialization and error handling also web request start and authenticate request.

Viewmodels

All the viewmodels are named Entity+Input, there is a base Input class which has the Id property. Also there are lots of attributes defined on the properties, they are used for validation and for the MUI. The int and DateTime properties from entities have a corresponding nullable property defined here (int?, DateTime?), Nullable is used because this way we don't get 0 and 1/01/0001 as default values in textboxes, also 0 as selected key for dropdowns.

User Interface

On all the pages where we have crud functionality we use the Awesome Grid, sometimes with CustomRender which makes it look more like a list. And on some pages we do the Create/Edit using popups while on others using the Grid's inline editing functionality.

All our crud demos are based on these demos:

https://demo.aspnetawesome.com/GridCrudDemo

https://demo.aspnetawesome.com/GridInlineEditDemo

The difference is that instead of just a Delete button, when the user is logged in as admin we need to show a restore button instead of Delete if the item is deleted. So because the value of the grid cell depends on the value of the row model (specifically IsDeleted property) and we didn't want to generate the html on the server side and pass it in the row model, we used ClientFormatFunc and we generate the html in js. You can see in ProUtils.DeleteColumn and ProUtils.InlineDeleteColumn will reference a js function (`ClientFormatFunc`) in proUtils.js to render the html.

The delete button will call the awe.open for the PopupForm that was initialized in the view using ProUtils.InitCrud or CrudHelpers.InitDeletePopupForGrid.
The restore button will call the awe.oc for the call initialized using the InitCall html helper in the View or inside the ProUtils.InitCrud.