

# ProDinner – asp.net mvc sample application

---

*Shows the pro way of developing an asp.net mvc web application.*

Keep the prodinner solution open while reading this so that you could read and look at the source.

## Project structure:

- Core – poco entities and interfaces for repositories and services
- Data – data access layer, contains repositories implementations, EF Code First mappings and the DbContext
- Service – service implementations
- Infra – contains the IoC related stuff
- Resources – the resource files used for Multi-Language user interface.
- WebUI – all the ViewModels and the Mapper classes which are used to map from Entity to Input (ViewModel) and backwards, mvc controllers, views, images, scripts and some bootstrapping code
- Tests – here I test the EF mappings, the controllers, and the valueinjections which are used for mapping entities to viewmodels and backwards

## Data Access

For data access I'm using Entity Framework Code First approach which presumes that I'm going to have clean POCO entities. All the properties in my entities are named exactly the same as the column names in my DB. Also the relationships are made accordingly to the EF default convention:

- for one-to-many I use **public virtual Entity PropName { get; set; }** together with **public int PropNameId { get; set; }** (which is the actual column name), this way I always have the Id and the actual object is loaded lazily when needed

*virtual for EF means that this property uses lazy-loading so it's not going to get filled until it's needed*

- for many-to-many **public virtual ICollection<Entity> Entities { get; set; }** on both sides, and I also do a bit of configuration for this in the Db class.

### Db

The **Db class** which inherits DbContext has a DbSet<Entity> for each entity as required by EF Code First, and I also override the OnModelCreating in order to configure the many-to-many relationships.

EF looks for a connection string name "Db" exactly as the class name of my DbContext.

### DbContextFactory

The DbContextFactory class is used to construct and get the DbContext, it has an interface which is resolved by the IoC **per-web-request**. So for each web request a new DbContext is going to be used and shared for all the repositories operations just for that web request.

### Repo<T>

Repo<T> is a generic repository which does all the basic DA actions. The Delete method doesn't delete entity but it sets the Property IsDeleted to true and saves it if the Entity implements IDel. The Where method gets all the entities according to the specified predicate (lambda) and also it can add one more condition which is IsDeleted = false if the Entity implements IDel and the showDeleted method parameter is false.

DelRepo<T> is used by the Repo<T> for actions specific to entities that implement IDel.

UniRepo is an universal repository which does some basic DA for any entity, the entity type is specified in the method call (this repo is used for mapping test where there is a single method that test all the entities automatically Tests.MappingTest AutoTest() method).

## Service Layer

### CrudService<TEntity>

The CrudService does all the CRUD operations, which basically call the same method in the repository and after repo.Save() which commits the changes made.

### UserService

Besides the functionality of the crudservice it also has its own method:

- IsUnique(login) - checks if this login is already in use, this used for validation for creating user, there is a LoginUniqueAttribute that is applied on the UserCreateInput.Login, this attribute uses this method
- Get(Login, Password) – tries to get a user by its login after compares the password string to the encrypted hash from Db, it returns null in case the login doesn't exist or the passwords didn't match
- Create is overridden in order to encrypt user's password before saving
- ChangePassword(id, password) – changes the password of the user with specified id

This class uses a Hasher class from the Encrypto library, this class is used to hash and salt the passwords before saving them, it is also used to compare a user input password to an encrypted password.

### FileManagerService

It is used to crop, save and delete images, it also gives random unique names to the new files (guid).

- SaveTempJpeg - resizes and saves the image from the inputstream into a temporary directory.
- MakeImages - takes the image from temp dir and makes 3 images of different sizes and saves them into meals dir.
- DeleteImages – is used to delete the old files when changing an image for a meal

### MealService

Inherits the functionality of CrudService<Meal> and also adds a method "SetPicture" that changes the picture for a meal on the disk and its name in the db.

## Infra

### IoC

The IoC class basically is the Service Locator (keeps an instance of the container and has methods for resolving types).

The WindsorRegistrar is used for registering types into the IoC. The RegisterAllFromAssemblies method takes the string name of an assembly (e.g. "Omu.ProDinner.Data") and registers all the implementations to the interface that they implement. There is also a Register method used to register a single type. Both these methods use PerWebRequest lifestyle and there is also a RegisterSingleton method which I use for Testing.

## Multilanguage User Interface

It's done using resource files. In the Resources project I have a resx file for each supported language, the file Mui.resx is used for English and all the other languages which are not supported, in rest we have a resx named Mui.(the language abbreviation).resx for each language. For all the resource files the access modifier is set to public so that they could be accessed from other projects (WebUI, Infra).

In Global.asax.cs Application\_BeginRequest we are creating a webforms fake page and set its culture, without this the asp.net is not gonna know which culture to use.

The functionality for changing the UI language is located in MuiController where the value for "lang" Cookie is set which is read in the Application\_BeginRequest, so this way the language from cookie is set at the beginning of each request.

## Authentication

Forms Authentication is used for this purpose, in the web.config it is specified mode="Forms" and the LoginUrl. There is an interface in Core.Security IFormsAuthentication that is implemented by FormAuthService from WebUI. This interface is used in AccountController for signing in/off users.

The roles of the users are being stored in the authentication cookie. In global.asax.cs Application\_AuthenticateRequest roles from the cookie are read and set into the current user.

## 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.

Here we have the IMapper<TEntity,TInput> interface and its implementations.

### Mapper<TEntity,TInput>

The purpose of this Mapper is to map from Entity to Input (ViewModel) and back. The mapper has 2 methods:

- MapToInput(TEntity) - builds an Input from the Entity
- MapToEntity(TInput, TEntity) - builds an Entity from an Input, it takes the TEntity and refills it's values with the corresponding ones from TInput

The MapToInput and MapToEntity methods use ValueInjector to fill the new/existing object. The default extension .InjectFrom(o) does all the properties which are of the same name and type in the ViewModel and Entity, for the rest there are some custom valueinjections that do from ICollection<Entity> to/from IEnumerable<int> and from nullables to/from normal types.

So this way almost all the mapping between entities and viewmodels are handled by this single class Mapper<TEntity, TInput> and **you can add more entities and viewmodels without the need to add more mapping code** because the generic mapping code (done with valueinjections) doesn't care about the types it maps, instead it just maps from any object to any object by matching:

- properties of the same name and type
- ICollection<AnyClassThatInheritsEntity> to IEnumerable<int> (from entity to viewmodel)
- IEnumerable<int> to ICollection<AnyClassThatInheritsEntity> (from viewmodel to entity)
- Normal to/from nullables (e.g. int <-> int?)

Also you can always inherit this Mapper and add some very custom mapping code by overriding the ToInput and ToEntity methods, or you can add some additional valueinjections to the Mapper.

### DinnerMapper

Inherits the generic Mapper<TEntity,TInput> which does the basic mapping and adds some additional custom mapping code. In WindsorConfigurator the IMapper<Dinner,DinnerInput> is registered to DinnerMapper so that the IoC container will use this class instead of the generic one for resolving the IMapper<Dinner,DinnerInput> type (used in DinnerController).

### Viewmodels

They are located in WebUI.Dto. All my viewmodels are named Entity+Input, I have 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?), I use nullable because this way I'm not going to get 0 and 1/01/0001 as default values in textboxes, also 0 as selected key for dropdowns.

## Cruder<TEntity, TInput>

This generic controller does the crud job for all the entities of the app, it makes use of the IMapper<TEntity, TInput> for mapping between entities and viewmodels and ICrudService<TEntity> for create/update/delete/get.

Cruder inherits the Crudere<TEntity, TCreateInput, TEditInput> and just overrides the EditView property so that one view will be used for both create and edit. The difference between Cruder and Crudere is that Crudere uses different viewmodels and views for edit and create. Its methods can be overridden in case anything custom is required for a certain action.

Generic controllers can't be used directly so I inherit the Cruder for each entity, and I also add some additional actions if needed.

## Bootstrapper

Has a single method called "Bootstrap" and it is used in Global.asax.cs Application\_Start() and it initializes the routes, it sets the controller factory, configures the Windsor IoC Container, and sets some default Settings for the awesome lib.

## Worker

The worker class is doing work in background. It is started in Bootstrapper using its start method and the purpose of it is to undelete dinners, chefs and meals periodically.

## User Interface

All Ajax helpers (AjaxList, Lookup, MultiLookup, Popup, etc.) are from ASP.net MVC Awesome, you can learn how they work and how to use them at projects home <http://ASPnetAwesome.com>, there is also sample applications available.

For **displaying** a list of the items the Html.Awe().AjaxList and Html.Awe().Grid helpers are used, the ajaxlist will render the items by calling this.RenderView (awesome extension method) and use a view from Views/Shared/ListItem e.g. Country.cshtml

For **creation** of a new item like Dinner, Country etc. there's a button generated using the Html.Awe().PopupForm helper that will call via ajax the Create action on the controller and open a popup that will contain the returned view (html), after clicking ok/save on the popup and the item is successfully saved, the popup closes and the specified success js function is called (e.g. createDinner, createMeal; located in Site.js), those functions will update the list of items, they will receive a parameter that contains the html for the new item, that parameter was returned by the HttpPost Create action (Crudere.Create)

**Editing** is almost the same as the creation except instead of 1 create button there's a button for each item in the list, you can see these buttons declared in View/Shared/ListItem/[Country.cshtml, Dinner.cshtml etc.]. The HttpPost Action (Crudere.Edit) besides the new rendered html of the item will also return the item's Id and the item Type.Name; the success js function (function edit in Site.js) specified in the Html.Awe().PopupForm declaration will find the existing item with the id equal to type+id and replace it with the new html.

**Deleteting** and **Restoring** is done by adding an html form with a button to each list item, depending on the `IsDeleted` property of the item the delete or the restore form is rendered. In the index view (e.g. `Meal/index.cshtml`) the `Html.Awe.Form` helper is declared and it will intercept the form post, add confirmation in case of deletion and send the form post via ajax, when it gets the response from the server it will execute a js success function, "del" for deletion success and "edit" for restore, both found in `Site.js`.

The Country Lookup in the create dinner view has CRUD functionality. The `CountryId` property in `DinnerInput` has `[AdditionalMetadata("CustomSearch", true)]` (because in the View the `EditorFor(o => o.CountryId)` helper is used, othewise you could do `Lookup(o => o.Country).CustomSearch(true)` in the view), this makes the lookup use a custom Search Form that is returned by `LookupController`'s action `SearchForm` (`CountryIdLookupController.SearchForm`) and in `SearchForm.cshtml` besides the search inputs there is a create button made using the `Url.Awe().PopupFormAction`. As for edit/delete/restore this lookup is rendering the same view (`Views/Shared/ListItemCountry.cshtml`) as the Country AjaxList, and in the `SearchForm.cshtml` there are `Html.Awe().Form` helpers declared that add confirmation and ajax functionality to the edit/delete forms so it works the same as in all the main pages except this is inside the lookup popup.