

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 Entity Framework Code First approach is being used which presumes that there's going to be clean POCO entities. All the entity properties are named exactly the same as the column names in the DB. Also the relationships are made accordingly to the EF default convention:

- for one-to-many use **public virtual Entity PropName { get; set; }** together with **public int PropNameId { get; set; }** (which is the actual column name), this way we 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 there's also 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 also override the OnModelCreating is overridden in order to configure the many-to-many relationships.

EF looks for a connection string named "Db" exactly as the class name of 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

[Microsoft.Owin](#) is used for this, (in the previous versions Forms Authentication was being used).

In packages.config you can see all the owin packages that need to be installed for this.

The configuration is found in App_Start/OwinConfig. In AccountController there's a private method SignInOwin which does the authentication work, creates the ClaimsIdentity, adds the roles, and signs in the 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.

ProMapper

This is a wrapper for the valueinjecter's `Mapper.Map<TSource,TResult>` function, we are using this method instead of just `Mapper.Map<TResult>` because EF will wrap the entity (source) in a proxy type and we won't have a map specified for the `<TProxy, TResult>`.

The mapper is configured in the **MapperConfig** class where we replace the default map method (`Mapper.DefaultMap`) to use an existing object as result instead of creating a new one, (that's because we want to get the object from EF and modify it using the mapper) and we also are using injections to map by convention (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 ProMapper and **you can add more entities and viewmodels without the need to add more mapping code**, so the DefaultMapper (`Mapper.Map`) will map this regardless of source and target type:

- 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?`)

We can also specify a map for each pair of types, this has been done for the `<Dinner, DinnerInput>` pair, where we still use the `DefaultMap` and after add some custom mapping code.

Viewmodels

They are located in `WebUI.Dto`. 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.

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 Cruder is inherited for each entity, and some additional

actions are added where 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 (Grid, 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 demo applications available.

For **displaying** a list of the items the awesome Grid and AjaxList helpers are used. The ajaxlist uses custom item template, achieved by rendering a view on the server, you can see the call to this.RenderView in [DinnersAjaxListController.Search](#).

For CRUD operations the Grid/AjaxList is used in combination with the PopupForm. Restoring items is done using the grid api for the grid (api.update) and using the Form helper for the ajaxlist.

The grid CRUD model is based on this demo: <http://demo.aspnetawesome.com/GridCrudDemo>

There's a PopupForm initialized for create, edit and delete item, each PopupForm has a js function (js functions are in Site.js) assigned that will execute when the form in the popup is successfully submitted, they will add the row, update it or remove with a small highlight/fade animation to emphasize the action. The js functions use the grid api for this and they use the gridmodel item (or just the id in case of delete) that was returned by the Crudere in the post action. When working with ajaxlist there's an additional parameter being set to the InitPopupForm helper (usingAjaxList) which tells Crudere not to return a gridmodel item but to render an { Id, Content } used to update the ajaxlist.

The delete PopupForm receives the gridid (html id) as a parameter and it's using it in ConfirmDelete.cshtml to emphasize (color yellow) the row that is about to be deleted.

The grid delete/restore cell is rendered using the deleteFormat js function which depending on the state of the item will render the delete button that opens the delete PopupForm, or restore button that calls the grid.api.update(id, params), and in the grid action (e.g. [CountryController/GridGetItems](#)) there's a check for a restore parameter if it exist and the current user is admin the restore will happen. For the AjaxList restore is done using the Html.Awe().InitCall helper; a call to the restore action is initialized using the helper and when the restore button is clicked the call is executed using a js function.

The Country Lookup in the create dinner view has CRUD functionality. You can see this in detail here: <http://demo.aspnetawesome.com/CrudInLookup>.

The CountryId property in DinnerInput has [[AdditionalMetadata](#)("CustomSearch", true)] (because in the

View the `EditorFor(o => o.CountryId)` helper is used, otherwise 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.Action().PopupFormAction`. As for edit/delete/restore this lookup is rendering the view from `Views/Shared/ListItem/Country.cshtml` and in the `SearchForm.cshtml` there's `InitPopupForm` 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.

If you have questions about prodinner you can ask them here:

<https://prodinner.codeplex.com/discussions>