**Naked Objects 8.0.0-beta –   
Application Developer Manual**

Contents

[Getting started 1](#_Toc456014756)

[Introduction to NOF 8 2](#_Toc456014757)

[Running the Template project 3](#_Toc456014758)

[Exploring the Example Model 4](#_Toc456014759)

[Upgrading an existing application 6](#_Toc456014760)

[Writing a new application 7](#_Toc456014761)

[Configuring the user interface 8](#_Toc456014762)

[Configuring colours for objects in the Gemini user interface 8](#_Toc456014763)

[Configuring error handling 9](#_Toc456014764)

[Configuring masks 9](#_Toc456014765)

[Other configuration options 10](#_Toc456014766)

[Adding user authentication 11](#_Toc456014767)

[NOF 8 user interface guide 12](#_Toc456014768)

[The Gemini user interface 13](#_Toc456014769)

[The Cicero user interface 14](#_Toc456014770)

[Basic operation 14](#_Toc456014771)

[Full list of Cicero commands 14](#_Toc456014772)

[Example use case 18](#_Toc456014773)

[Programming Reference 19](#_Toc456014774)

[Domain model - programming concepts 20](#_Toc456014775)

[Domain object 20](#_Toc456014776)

[Property 20](#_Toc456014777)

[Action 21](#_Toc456014778)

[Menus 22](#_Toc456014779)

[Recognised method 24](#_Toc456014780)

[Recognised attribute 25](#_Toc456014781)

[View Model 25](#_Toc456014782)

[Service 28](#_Toc456014783)

[Factories and Repositories 28](#_Toc456014784)

[External or System service 30](#_Toc456014785)

[Contributed action 31](#_Toc456014786)

[Injection of domain services into domain objects 33](#_Toc456014787)

[The Domain Object Container 34](#_Toc456014788)

[Application configuration 37](#_Toc456014789)

[Specifying the Namespace(s) that cover your domain model 37](#_Toc456014790)

[Registering domain services 37](#_Toc456014791)

[Specifying any types that will not ordinarily be discovered by the reflector 38](#_Toc456014792)

[Configuring the EntityObjectStore 39](#_Toc456014793)

[Configuring Authorization 39](#_Toc456014794)

[Configuring Auditing 40](#_Toc456014795)

[Configuring Profiling 40](#_Toc456014796)

[Configuring the RestRoot 41](#_Toc456014797)

[System configuration using the Unity framework 42](#_Toc456014798)

[The Naked Objects programming model 43](#_Toc456014799)

[Recognised Value Types 43](#_Toc456014800)

[Recognised Collection types 43](#_Toc456014801)

[Recognised .NET attributes 44](#_Toc456014802)

[Recognised Methods 47](#_Toc456014803)

[NakedObjects.Attributes 50](#_Toc456014804)

[NakedObjects.Types 59](#_Toc456014805)

[NakedObjects.Helpers 60](#_Toc456014806)

[A how-to guide 63](#_Toc456014807)

[Using the Naked Objects IDE 64](#_Toc456014808)

[Item Templates 64](#_Toc456014809)

[Code Snippets 64](#_Toc456014810)

[The object life-cycle 67](#_Toc456014811)

[How to create an object 67](#_Toc456014812)

[How to persist an object 68](#_Toc456014813)

[How to update an object 68](#_Toc456014814)

[How to delete an object 68](#_Toc456014815)

[How to retrieve existing instances 69](#_Toc456014816)

[How to insert behaviour into the object life cycle 69](#_Toc456014817)

[How to specify that an object should never be persisted 69](#_Toc456014818)

[How to specify that an object should not be modified by the user 69](#_Toc456014819)

[How to specify that a class of objects has a limited number of instances 70](#_Toc456014820)

[How to implement concurrency checking 70](#_Toc456014821)

[Object presentation 71](#_Toc456014822)

[How to specify a title for an object 71](#_Toc456014823)

[How to control the layout of an object’s action menu 72](#_Toc456014824)

[How to specify a name and/or description for an object 72](#_Toc456014825)

[How to specify that an object should be always hidden from the user 72](#_Toc456014826)

[Properties 73](#_Toc456014827)

[How to add a property to a domain object 73](#_Toc456014828)

[How to prevent the user from modifying a property 73](#_Toc456014829)

[How to make a property optional (when saving an object) 74](#_Toc456014830)

[How to specify the size of String properties 74](#_Toc456014831)

[How to validate user input to a property 74](#_Toc456014832)

[How to validate user input to more than one property 75](#_Toc456014833)

[How to specify a default value for a property 76](#_Toc456014834)

[How to specify a set of choices for a property 77](#_Toc456014835)

[How to specify auto-complete for a property 78](#_Toc456014836)

[How to set up the initial value of a property programmatically 79](#_Toc456014837)

[How to trigger other behaviour when a property is changed 79](#_Toc456014838)

[How to control the order in which properties are displayed 80](#_Toc456014839)

[How to specify a name and/or description for a property 80](#_Toc456014840)

[How to hide a property from the user 80](#_Toc456014841)

[How to make a property non-persisted 81](#_Toc456014842)

[How to handle File Attachments 81](#_Toc456014843)

[How to display an image 83](#_Toc456014844)

[How to handle enum properties 84](#_Toc456014845)

[How to work with date properties 85](#_Toc456014846)

[Collection properties 86](#_Toc456014847)

[How to add a collection property to a domain object 86](#_Toc456014848)

[Adding-to or removing objects from a collection 86](#_Toc456014849)

[How to create a derived collection 87](#_Toc456014850)

[How to control the order in which table rows are displayed 88](#_Toc456014851)

[How to specify which columns are displayed in a table view 89](#_Toc456014852)

[Actions 90](#_Toc456014853)

[How to add an action to an object 90](#_Toc456014854)

[How to specify the layout of the menu of actions on an object 90](#_Toc456014855)

[How to define a contributed action 90](#_Toc456014856)

[How to prevent a service action from being a contributed to objects 90](#_Toc456014857)

[Hor to specify parameter names and/or descriptions 90](#_Toc456014858)

[How to make a parameter optional 90](#_Toc456014859)

[How to specify a default value for a parameter 91](#_Toc456014860)

[How to specify a set of choices for a parameter 91](#_Toc456014861)

[How to allow selection of multiple choices 93](#_Toc456014862)

[How to specify auto-complete for a parameter 93](#_Toc456014863)

[How to specify the length or format for text-input parameters 94](#_Toc456014864)

[How to obscure input text (e.g. for a Password) 94](#_Toc456014865)

[How to validate parameter values 94](#_Toc456014866)

[How to work with date parameters 95](#_Toc456014867)

[How to specify conditions for invoking an action 96](#_Toc456014868)

[How to control the order in which actions appear on the menu 97](#_Toc456014869)

[How to hide actions 97](#_Toc456014870)

[How to pass a message back to the user 98](#_Toc456014871)

[How to work with transactions 98](#_Toc456014872)

[Working with Entity Framework 99](#_Toc456014873)

[Overriding the default database schema generation 99](#_Toc456014874)

[Using data fixtures with Code First 100](#_Toc456014875)

[How to handle concurrency checking 101](#_Toc456014876)

[How to specify 'eager loading' of an object's reference properties 101](#_Toc456014877)

[How to implement complex types 102](#_Toc456014878)

[How to work with multiple databases 102](#_Toc456014879)

[How to work with multiple database contexts 102](#_Toc456014880)

[How to write safe LINQ queries 104](#_Toc456014881)

[How to handle associations that are defined by an interface rather than a class 106](#_Toc456014882)

[Customising the Restful API 113](#_Toc456014883)

[Adding custom views to the UI 123](#_Toc456014884)

[Layout of properties 124](#_Toc456014885)

[Render an action as a top-level button 125](#_Toc456014886)

[Polling for updates 126](#_Toc456014887)

[Other 127](#_Toc456014888)

[Getting hold of the current user programmatically 127](#_Toc456014889)

[Creating an XML Snapshot of an object 128](#_Toc456014890)

[How to run Naked Objects as a .exe 130](#_Toc456014891)

[How to run multiple threads asynchronously 131](#_Toc456014892)

[Authorization 132](#_Toc456014893)

[Attribute-based Authorization 132](#_Toc456014894)

[Custom Authorization 134](#_Toc456014895)

[Adding further security measures 135](#_Toc456014896)

[Auditing 137](#_Toc456014897)

[Profiling 138](#_Toc456014898)

[Testing 139](#_Toc456014899)

[Executable Application Tests (XATs) 139](#_Toc456014900)

[End to End Testing with Selenium [TBD. Is this still valid?] 143](#_Toc456014901)

[How to use the cluster pattern 145](#_Toc456014902)

[The Clusters project on GitHub 148](#_Toc456014903)

[How to build the framework from source 149](#_Toc456014904)

[Troubleshooting 150](#_Toc456014905)

[Logging 151](#_Toc456014906)

[Errors thrown when starting an application 152](#_Toc456014907)

[No known services 152](#_Toc456014908)

[Unable to infer a key 152](#_Toc456014909)

[Class not public 152](#_Toc456014910)

[Errors thrown when running an application 153](#_Toc456014911)

[A property is not virtual/overrideable 153](#_Toc456014912)

[Invalid column name 153](#_Toc456014913)

[Invalid object name 153](#_Toc456014914)

[Collection not initialised 153](#_Toc456014915)

[Database is generated, but certain (or all) tables are not being generated 153](#_Toc456014916)

[Unexpected behaviour in the user interface 155](#_Toc456014917)

[A public method is not appearing as an object action 155](#_Toc456014918)

[Default, Choices, Validate or other complementary methods are showing up as menu actions 155](#_Toc456014919)

[Debugging 156](#_Toc456014920)

[Life Cycle methods are not being called 156](#_Toc456014921)

[The injected Container is null 156](#_Toc456014922)

[An injected service is null 156](#_Toc456014923)

## 

# Getting started

## Introduction to NOF 8

Version 8.0 of the Naked Objects Framework (NOF 8) offers a brand new user interface, not just a very different look and feel, but a brand new ‘Single Page’ architecture (SPA). The client, written in TypeScript compiled down to JavaScript, and using the Angular.js framework (version 1.5), provides a great deal of locally-executed functionality. It communicates with the server via a Restful API, that conforms to the Restful Objects 1.1 public specification (see [www.restfulobjects.org](http://www.restfulobjects.org) ).

## Running the Template project

The best way to start using NOF8 for the first time, or to add a NOF8 run project to an existing NOF7 application, is to use the NakedObjects.Template project. You can download this as a .zip file from here: <https://github.com/NakedObjectsGroup/NakedObjectsFramework/blob/master/Run/NakedObjects.Template.zip?raw=true>.

**Note**: We recommend using the .zipped version of the Template project, rather than taking the source directly from the Master branch of the repository - because the latter is under continusous development and may references versions of packages that have not yet been released publicly on the NuGet gallery.

It should be possible to just run the unzipped project ‘as is’ within Visual Studio 2015 - this starts the server, and the single page application to connect with it.

**Troubleshooting**: If, when you start up the application it displays the message ‘Failed to connect to server …’ this usually means that the server has not started up. You can check the log file nakedobjects\_log.txt (found in the same directory as the project file) for any errors. The most likely cause is that Entity Framework has been unable to create the database. Check the connection string in Web.config:

<connectionStrings>

<add name="NakedObjectsExample"

connectionString="Data Source=(LocalDB)\MSSQLLocalDB;

Initial Catalog=NakedObjectsExample;

Integrated Security=True;

MultipleActiveResultSets=True"

providerName="System.Data.SqlClient" />

</connectionStrings>

this is set to work with LocalDB, which should be set up by default on VS2015. If you do not have LocalDB set up, try changing the connection string to point to a database that does work for you e.g. SQLEXPRESS.

The Template project contains a *minimal* [example model](#_Exploring_the_Example). Running the project will create the database . You should then be able to create a Customer, and retrieve all saved customers. See also [NOF 8 user interface guide](#_NOF_8_user).

Once you have successfully run it. Note the following:

1. The Template application has installed the NakedObjects.Run NuGet package. This in turn has installed various dependent Naked Objects packages.
2. The [example model](#_Exploring_the_Example) is included within the ‘run’ project. In a real application we recommend that you create your domain object model in one or more separate projects. The model project(s) would then depend only on the very light-weight NakedObjects.ProgrammingModel package - not the rest of the Naked Objects framework.
3. Within the run project, the application is configured primarly through the NakedObjectsRunSettings class. For details of what must be specified see: [Application configuration](#persistor). (For existing users: note that this is exactly the same as for a NOF 7 run project).
4. The template run project creates the server that provides a complete Restful API from the domain model, and also provides the ‘single page application’ client that runs within a browser. The latter consists of the single html page (index.html) and the necessary script files, some of which are specific to Naked Objects and others being third party frameworks such as Angular, LoDash, and JQuery. All of the Naked Objects scripts are written in TypeScript and compiled to JavaScript.
5. For a larger scale application you might prefer to maintain the Restful API server and the Spa client in separate projects. To do this you should un-install just the NakedObjects.Run and NakedObjects.Spa packages, leaving all the other Naked Objects packages installed. Then you can install just the NakedObjects.Spa package into a new ASP.NET project, and [configure](#_Other_configuration_options_1) the nakedbjects.config.ts script to point to the Restful API. Note, however, that you will probably also need to configure the server for cross-origin resource sharing (CORS) using standard ASP.NET patterns.

### Exploring the Example Model

The example model consists of two domain classes. Customer is a [domain object](#domain_object) (or entity) class:

public class Customer {

[NakedObjectsIgnore]

public virtual int Id { get; set; }

[Title]

public virtual string Name { get; set; }

}

Note that:

* *All* persisted properties in a Naked Objects application must be virtual.
* [NakedObjectsIgnore] specifies that this property is never going to be displayed on the user interface.
* [Title] specifes that the value of that property should be displayed as the [title](#object_title) for that object (shown at the top of a view, or in a link to that object).

CustomerRepository is a [domain service](#service) for creating and retrieving those entities (strictly speaking it is a combined ‘factory’ and ‘repository’ service – you can separate those two roles into two services if you prefer):

public class CustomerRepository {

public IDomainObjectContainer Container { set; protected get; }

public Customer CreateNewCustomer() {

return Container.NewTransientInstance<Customer>();

}

public IQueryable<Customer> AllCustomers() {

return Container.Instances<Customer>();

}

public IQueryable<Customer> FindCustomerByName(string name) {

return AllCustomers().Where(c => c.Name.ToUpper().Contains(name.ToUpper()));

}

}

The [IDomainObjectContainer](#_The_Domain_Object_1) is the single point of contact between domain model code and the outside world. Amongst other things, this Container acts as a wrapper for the Entity Framework, so your domain model need not (and should not) have any direct contact with Entity Framework functionality. The container is automatically injected into this repository by the Naked Objects Framework; the container may also be injected into an entity object; and you can use the same pattern to [inject an instance of your own services](#dependency_injection), such as this CustomerRepository, into other services or entities.

The domain model project includes responsibility for mapping the objects onto a database using standard Entity Framework Code First patterns - by means of the ExampleDbContext class:

public class ExampleDbContext : DbContext{

public ExampleDbContext(string dbName) : base(dbName) {

Database.SetInitializer<DbContext>(new   
 DropCreateDatabaseIfModelChanges<DbContext>());

}

public DbSet<Customer> Customers { get; set; }

}

In this example, the mapping from the domain type to the database tables is automatic, following default conventions; in a real project you might want to take direct control of the mapping, using standard Entity Framework patterns. You might also prefer to extract the DbContext, mapping and data fixture classes into a separate project, from the domain model. For more detail see [Working with Entity Framework](#_Working_with_Entity).

## Upgrading an existing application

Upgrading an existing project from NOF7 to NOF8 is straightforward – at least if you are using the generic Naked Objects user interface. However, any customisations you have made to the generic UI will need to be completely re-written to work in the new Single Page Application client architecture, and you will need to be familiar with TypeScript and Angular to do this. See [Adding custom views to the UI](#_Adding_custom_views).

**Note**: we do not recommend attempting to upgrade directly to NOF8 from NOF6 or earlier versions of the framework, without having first upgraded to NOF7 and run successfully. (Instructions for upgrading from NOF6 to NOF7 were included in the NOF7 Developer Manual, which may be obtained from the .zip of the NOF7.0 release on GitHub).

If you have an existing application running under NOF 7, then the good news is that *no changes are required to your domain model(s)*. NOF 8.0 has deliberately stayed with the 7.0 Programming Model for this reason. As well as minimising the upgrade effort, the huge advantage of this design decision is that you can run both the NOF 7 client and the NOF 8 client in parallel, against the same domain model(s). Indeed, we recommend running the two in parallel *rather than upgrading your existing Run (client) project(s)*, at least until NOF 8 has gone to full release and you are happy with the new user interface.

1. So start by obtaining the [NakedObjects.Template](#_Getting_started) project and add this to your solution alongside your existing NOF7 Run project.
2. Rename the project and namespaces as desired.
3. Add references to your domain model project(s).
4. Delete the Example Model folder.
5. Replace the NakedObjectsRunSettings from the template with a copy of the equivalent file in your NOF7 run project.

## Writing a new application

We recommend that you create your domain model in separate project(s). A domain model project needs to have the NakedObjects.ProgrammingModel NuGet package installed, but not any other part of the Naked Objects Framework.

Before proceding further we recommend that you install the NakedObjects.Ide NuGet package. This does not contain any run-time code, and is therefore *not* necessary for writing a Naked Objects application, but it does contain a number of useful file templates and code snippets, that can make writing applications even faster. It is not necessary to have this package installed in each project. Once you’ve installed it into any project, the templates and code snippets will be available on all Visual Studio projects.

Now start to create your domain model, following the patterns and conventions described in [Domain model - programming concepts](#_Domain_model_-) and [A how-to guide](#_A_how-to_guide), and using the short-cuts described in See [Using the Naked Objects IDE](#object_lifecycle).

To turn your domain model into an application you will need to add a ‘Run’ project into your solution, and add references to the domain model. By far the best way to so this is to use a copy of [Template project](#_Getting_started_with), changing the project name and namespaces as desired. Delete the Example Model folder and all references to it in NakedObjectsRunSettings.

Now configure the application, following the guidelines in [Application configuration](#persistor) .

## Configuring the user interface

### Configuring colours for objects in the Gemini user interface

The colours for individual object types are determined by the rules specified in the file nakedobjects.services.color.config.ts within the method

module NakedObjects {

app.run((color: IColor) => {

//Add colour rules in here  
 }  
}

The rules are executed at run-time, and the results are cached in a dictionary. There are four types of rule as described below.

You may specify the colour for a single object type (given its fully-qualified name), for example:

color.addType("AdventureWorksModel.SpecialOffer", 5);

Note that all colours are specified as numbers. The numbers are translated into actual colours by the .css (described below).

You may specify the colour for any object type whose fully-qualified name matches a given RegEx pattern, for example:

const matchBusiness = /AdventureWorksModel.Business\*/;

color.addMatch(matchBusiness, 8);

You may specify the colour for any sub-type of a given type (which may be a class or an interface), for example:

color.addSubtype("AdventureWorksModel.BusinessEntity", 6);

Note: matching sub-types is a relatively expensive mechanism in terms of round-trips to the server (though the results are cached), so it should be used sparingly.

You may specify a default colour, for example:

color.setDefault(0);

The rules may be added in any order, but they are applied in the order in which they are written, and the colour for a given object type will be determined by the first rule that matches the type. It follows that the default colour should be specified last.

The colour numbers are interpreted by the .css file, for example:

.object-color0 {

background-color: #000000; /\*Black\*/

}

.link-color0 {

background-color: #303030 /\*#525252\*/ ; /\*Dark grey\*/

}

.object-color1 {

background-color: #2b5797; /\*Blue\*/

}

.link-color1 {

background-color: #1b4787; /\*Slightly-lighter blue\*/

}

For each colour number there are two variants: the .object-color and the .link-color. The first is used as the background for a view of that type of object, and the second it used as the background for a link to an object. The recommended approach is to keep these two colours similar, but with a slight difference in brightness. This is so that a link will always show up against a background that has the same colour number.

### Configuring error handling

The file nakedobjects.services.error.config.ts allows you to configure the way that specific types of server errors are handled. In the default configuration, for example, the following code specifies that if the error is a concurrency error, then the concurrencyTemplate should be applied to the display…

error.setErrorDisplayHandler(($scope: INakedObjectsScope) => {

if ($scope.error.isConcurrencyError) {

$scope.errorTemplate = concurrencyTemplate;

}

});

The concurrencyTemplate is defined in nakedobjects.constants.ts as this file:

"Content/partials/concurrencyError.html";

You can add your own templates for specific errors.

### Configuring masks

Masks, for formatting value strings, may be specified in the domain model, using the [Mask](#d0e5093) attribute. However, the interpretation of these masks is the responsibility of the client, and this is specified in nakedobjects.services.mask.config.ts. The default specifications for recognised masks are as follows:

mask.setCurrencyMaskMapping("C", "decimal", pound, 2);

mask.setCurrencyMaskMapping("c", "decimal", pound, 2);

mask.setDateMaskMapping("d", "date-time", "d MMM yyyy", "+0000");

mask.setDateMaskMapping("D", "date", "d MMM yyyy hh:mm:ss");

mask.setDateMaskMapping("T", "time", "HH:mm", "+0000");

For example, [Mask("d")] will present only the date part of a DateTime value, and [Mask("c")] applied to a Decimal property will render it as a currency value.

**Note:**  The specification of the value types in the code above (for example: "decimal", "date-time") are the types as defined in the Restful Objects specification, which is the standard used for communication between the Naked Objects client and server.

### Other configuration options

Several other configuration options (besides colour) are specified in the nakedobjects.config.ts file for example as shown here

export var appPath = "http://localhost:61546";

export const defaultPageSize = 20; // can be overriden by server

export const listCacheSize = 5;

export const shortCutMarker = "\_\_\_";

export const urlShortCuts = ["http://myUrl", "myNamespace"];

export const keySeparator = "--";

export const objectColor = "object-color";

export const linkColor = "link-color";  
 export const autoLoadDirty = true;  
 export const showDirtyFlag = false || !autoLoadDirty;

The combination of the shortCutMarker and urlShortCuts are used simply to reduce the size of the Urls used to navigate within the Spa. (Note that these Urls are not sent to the server - those Urls all follow the format specified in the Restful Api). The urlShortCuts may be used to define any long-ish strings of characters that are likely to recur frequently within the generated Urls, including (but not limited to) namespaces used on domain model types (such as the example of AdventureWorksModel above) and/or the Url root of the Restful API for your server. At run-time, where a match is found for any of these strings in a generated Url, it will be compressed into the form \_\_\_0, \_\_\_1, etc, where \_\_\_ is as specified in the shortCutMarker and the number is the index of the matching string within the urlShortCuts array.

The keySeparator (-- by default) is used between the type and the Id value in any object identifier, and between each key if there are multiple. *You should not change this unless you make an equivalent change within the server providing the Restful API*.

The objectColour and linkColour constants define the classes added to the generated Html, to be interpreted by the .css. If you change them you will need to change the .css also.

The autoLoadDirty and showDirtyFlag constants provide options for trading off liveness against performance. NOF8 caches object data in the client. When an action is invoked on an object that object is marked, internally, with a ‘dirty’ flag - meaning that its data *might* have changed. Unfortunately, the client cannot know whether the action (executed on the server) *actually* changed data or not. The exception is if the action was a ‘query only’ action - *either* because it returned an IQueryable or because it is explicitly marked-up with the QueryOnly annotation – in which case the client assumes that there has been no change. With the autoLoadDirty flag set to true, if the user navigates back to that object it will be automatically reloaded from the server - to ensure it is up to date. If the flag is set to false, the user may hit the Reload button, if it is important to ensure that it is up-to-date. If the showDirtyFlag is then set to true (as it would be by default given the clause ‘|| !autoLoadDirty’) then any ‘dirty’ object will have an asterisk after its title. (Please note, again, that this asterisk does not mean that the data *is* out of date, just that it *might be*).

### Adding user authentication

1. Add the default MVC routes. We suggest creating a method like this, either added to the existing RestfulObjectsConfig class, or a new custom RouteConfig class:

public static void RegisterMvcRoutes(RouteCollection routes) {

. . .

routes.MapRoute(

"Default", "{controller}/{action}/{id}",

new { controller = "Account", action = "Login",  
 id = UrlParameter.Optional}

);

}

where the controller and action are specified as needed.

2. This RegisterMvcRoutes method must be called in RestfulObjectsActivator PreStart method.

3. Add the [Authorize] attribute to RestfulObjectsController.

# NOF 8 user interface guide

## The Gemini user interface

The default user interface for NOF 8 is known as Gemini. It has modern or ‘flat’ styling, and makes optional use of coloured backgrounds to help distinguish the most important types of object. Gemini runs in all modern popular browsers.

We strongly recommend that you run Gemini in full screen mode, since all the browser functions that you need are provided as icons at the bottom of the screen.

The Gemini name reflects the fact that it supports ‘twin panes’ – the screen may be split into two panes, allowing you to view and use two different objects, or lists of objects at once.

Gemini has been designed to use familiar modes of interation and should require no up-front training - just explore clicking on the obvious icons, buttons and links. However, two things may not be obvious:

* When clicking on an active area, such as an action, icon, or link (shown as a block of contrasting colour or tint), you have *in many cases* the option to click with the left or right mouse-buttons. The left (standard) button will open the next view in the same pane; the right mouse-button will open the next view in the *other* pane, splitting the screen if it is not already split.
* Gemini supports a limited form drag and drop. If you have a field in a dialog box, or an object in Edit view that requires a particular type of domain object, then you can drag and drop an object of the correct type from the other pane. You can either drag a ‘link’ (small coloured bar) from a list, or if you are looking at a view of a single object, you can drag that object by its title.

## The Cicero user interface

Cicero is a user interface purpose-designed to work with an audio screen-reader.

It can be accessed, from any context, by clicking on the speech-balloon icon at the bottom of the Gemini screen. If you do not wish to have this capability available to users, hide the icon by adding this line to the .css:

.icon-speech {display: none}

**Note:** Cicero is still experimental. While it is substantially usable, it is not 100% implemented as yet.

### Basic operation

The display has only two fields: a read-only output field, and a single input field. The input field always has the focus. Commands are typed into the input field followed by the Enter key.When the output field updates (either instantaneously or after the server has responded) its contents are read out automatically, so the user never has to navigate around the screen.

Commands, such as action, field and save, may be typed in full or abbreviated to the first two or more characters. Commands are not case sensitive.

Some commands take one or more arguments. There must be a space between the command word and the first argument, and a comma between arguments. Arguments may contain spaces if needed.

The commands available to the user vary according to the context. The command help ? (note that there is a space between help and ?) will list the commands available to the user in the current context. help followed by another command word (in full or abbreviated) will give more details about that command.

Some commands will change the context, for example using the go command to navigate to an associated object, in which case the new context will be read out. Other commands - help being an example - do not change the context, but will read out information to the user. If the user needs a reminder of the current context, the where command will read the context out again. Hitting the enter-key on the empty input field has the same effect.

When the user enters a command and the output has been updated, the input field will be cleared, ready for the next command. The user may recall the previous command, by hitting the up- arrow key. The user might then edit or extend that previous command and hit Enter to run it again.

For advanced users: commands may be chained using a semi-colon between them, however commands that do, or might, result in data updates cannot be chained.

### Full list of Cicero commands

#### action

Open the dialog for action from a menu, or from object actions. A dialog is always opened for an action, even if it has no fields (parameters). This is a safety mechanism, allowing the user to confirm that the action is the one intended. Once the dialog fields have been completed, using the enter command, the action may then be invoked with the ok command.

The action command takes two optional arguments. The first is the name, or partial name, of the action. If the partial name matches more than one action, a list of matches is returned but none opened. If no argument is provided, a full list of available action names is returned.

The partial name may have more than one clause, separated by spaces. These may match either parts of the action name or the sub-menu name if one exists. If the action name matches a single action, then a question-mark may be added as a second parameter, which will generate a more detailed description of the Action.

#### **back**

Move back to the previous context.

#### cancel

Leave the current activity (action dialog, or object edit), incomplete.

clipboard

The clipboard command is used for temporarily holding a reference to an object, so that it may be used later to enter into a field. clipboard requires one argument, which may take one of four values: copy, show, go, or discard, each of which may be abbreviated down to one character.

* copy copies a reference to the object being viewed into the clipboard, overwriting any existing reference.
* show displays the content of the clipboard without using it.
* go takes you directly to the object held in the clipboard.
* discard removes any existing reference from the clipboard.

The reference held in the clipboard may be used within the Enter command.

#### edit

Put an object into edit mode.

#### enter

Enter a value into a field, meaning a parameter in an action dialog,or a property on an object being edited. enter requires 2 arguments. The first argument is the partial field name, which must match a single field.

The second optional argument specifies the value, or selection, to be entered. If a question mark is provided as the second argument, the field will not be updated but further details will be provided about that input field.

If the word paste is used as the second argument, then, provided that the field is a reference field, the object reference in the clipboard will be pasted into the field.

#### forward

Move forward to next context in the history(if you have previously moved back).

#### gemini

Switch to the Gemini (graphical) user interface preserving the current context.

#### goto

Go to the object referenced in a property, or to a collection within an object, or to an object within an open list or collection.

goto takes one argument. In the context of an object that is the name or partial name of the property or collection. In the context of an open list or collection, it is the number of the item within the list or collection (starting at 1).

#### help

If no argument is specified, help provides a basic explanation of how to use Cicero.

If help is followed by a question mark as an argument, this lists the commands available in the current context. If help is followed by another command word as an argument (or an abbreviation of it), a description of the specified command is returned.

#### menu

Open a named main menu, from any context. menu takes one optional argument: the name, or partial name, of the menu. If the partial name matches more than one menu, a list of matches is returned but no menu is opened; if no argument is provided a list of all the menus is returned.

#### ok

Invoke the action currently open as a dialog. Fields in the dialog should be completed before this.

#### page

Supports paging of returned lists. The page command takes a single argument, which may be one of these four words: first, previous, next, or last, which may be abbreviated down to the first character. Alternative, the argument may be a specific page number.

#### reload

*Not yet implemented*. Reload the data from the server for an object or a list. Note that for a list, which was generated by an action, reload runs the action again, thus ensuring that the list is up to date. However, reloading a list does not reload the individual objects in that list, which may still be cached. Invoking Reload on an individual object, however, will ensure that its fields show the latest server data.

#### root

From within an opened collection context, the root command returns to the root object that owns the collection. Does not take any arguments.

#### save

Save the updated fields on an object that is being edited, and return from edit mode to a normal view of that object.

#### selection

*Not yet fully implemented*. Select one or more items from a list, prior to invoking an action on the selection. selection has one mandatory argument, which must be one of these words: add, remove, all, clear, show.

The add and remove options must be followed by a second argument specifying the item number, or range, to be added or removed.

#### show

In the context of an object, shows the name and content of one or more of the properties. May take 1 argument: the partial field name. If this matches more than one property, a list of matches is returned. If no argument is provided, the full list of properties is returned.

In the context of an opened object collection, or a list, shows one or more items from that collection or list. If no arguments are specified, show will list all of the the items in the opened object collection, or the first page of items if in a list view.Alternatively, the command may be specified with an item number,or a range such as 3-5.

#### where

Display a reminder of the current context. The same can also be achieved by hitting the Return key on the empty input field.

### Example use case

Out: Welcome to Cicero  
**In: help ?**Out: Commands available in current context: back, clipboard, forward, gemini, help, menu, where,  
**In: menu**Out: Menus: Customers, Orders, Products …  
**In: me cus**Out: Customers menu.  
**In: ac**Out: Actions: Find Customer By Account Number, Stores - Find Store By Name…  
**In: ac find indiv**Out: Customers menu. Action dialog: Find Individual Customer By Name. First Name: empty, Last Name: empty,  
**In: enter first, k**Out: Customers menu. Action dialog: Find Individual Customer By Name. First Name: k, Last Name: empty,  
**In: en last, henders**Customers menu. Action dialog: Find Individual Customer By Name. First Name: k, Last Name: henders,  
**In: ok**Out: Result from Find Individual Customer By Name: Page 1 of 1 containing 5 of 5 items  
**In: show 1,2**Out: Item 1: Kaitlyn Henderson, AW00012132; Item 2: Kelly Henderson, AW00012920;  
**In: go 2**Out: Customer: Kelly Henderson, AW00012920.  
**In: ac**Out: Actions: Create New Order, Quick Order, Search For Orders, Last Order…  
**In: ac last**Out: Customer: Kelly Henderson, AW00012920. Action dialog: Last Order.  
**In: ok**Out: Sales Order Header: SO66127.  
**In: enter date**Out: Order Date: 2008-03-13T00:00:00Z, Due Date: 2008-03-25T00:00:00Z, Ship Date: 2008-03-20T00:00:00Z, Modified Date: 2008-03-20T00:00:00Z,  
**In: go det**Out: Collection: Details on Sales Order Header: SO66127, 2 items  
**In: show**Out: Item 1: 1 x HL Mountain Tire; Item 2: 1 x Fender Set - Mountain;

# Programming Reference

## Domain model - programming concepts

This section describes the key concepts in a Naked Objects application, and how they fit together. Together with the section [Adding behaviour to your domain objects - a how-to guide](#_Adding_behaviour_to) this tells you all you need to know to create and deploy your own applications.

### Domain object

Domain objects represent the persistent entities of the domain, such as: customers, products and orders.

In Naked Objects, any 'Plain Old CLR Object' (POCO) can function as a domain object - in other words a domain class does not have to inherit from any special class, nor implement any particular interface, nor have any specific attributes. There are three important things to remember:

* All (persisted) [properties](#_Property) must be declared virtual.
* All [collections](#collection_property) must be initialised.
* Domain objects do not need an explicit constructor, and we recommend against writing constructors, because constructor logic may be executed before the framework has initialised the object properly. For more information about the lifecycle of domain objects see [The object life-cycle](#_The_object_life-cycle).

You can specify certain things about both the behaviour and presentation of domain objects by adding specific attributes or methods. See [Object presentation](#_Object_presentation).

### Property

For a property of a domain object to be recognised by Naked Objects, it must be public and virtual. There is no other special programming required. Use the [propv snippet](#_Snippets_for_creating) as a shortcut.

#### Value Property

A value property has a string, number, date, or other [recognised value type](#value_types). This will be rendered to the user as a textual field. Assuming that the user is allowed to modify that property, they may enter the value by typing in text, which will be validated and formatted according to the value type. (Certain value types may provide alternative mechanisms for user input, such as a calendar-selector for a date field.)

#### Reference Property

A reference property is one where the type is another domain object. Reference properties are thus sometimes referred to as 'associations'. This will be rendered on screen as a field containing the referenced object rendered as a link that the user may follow. In edit mode, the user may specify the object to be associated by a number of mechanisms such as: auto-complete, drag and drop, copy and paste, or selection from a drop-down list.

#### Collection Property

A collection property is a property that returns any of the [recognised collection types](#recognised_collection_types). However, it is recommended that you specifically use ICollection<T>. Collections must be initialised (but in-line, not in the constructor). Collection properties are not paged: if the contents of the collection property are displayed (for example in the form of a list or a table) then the full contents will be shown.

Warning: It is therefore recommended that you never have large collections represented as collection properties. In other words: do not represent associations that have large cardinality as being directly navigable. Where the number of associated objects may be large, then the navigation should be via action methods rather than via a collection property. This is recognised by many modellers as being good modelling practice - it is not just a constraint of Naked Objects.

You can specify certain things about both the behaviour and presentation of properties by adding specific attributes or methods. See [Properties](#_Properties) and [Collection properties](#_Collection_properties).

### Action

An action is a method that is intended to be invoked by a user - though it may also be invoked programmatically from within another method or another object.

By default, any public instance method that you add to a class (whether it is a domain class or a service) will be treated as a user action, provided that all its parameter types (if any) and its return type (if any) are types recognised by Naked Objects. A method will also not be treated as an action it represents a property or another [recognised method](#recognised_methods). Note also that static methods are ignored by Naked Objects.

Tip: Use the [Action snippet](#d0e5992) (shortcut act) to create an action method.

If you have a method that you don't want to be made available as a user-action you should either:

* give it a non-public access modifier
* Mark it with the NakedObjectsIgnore attribute.

You can specify certain things about both the behaviour and presentation of actions by adding specific attributes or methods. See [Actions](#_Actions).

See also: [Contributed action](#contributed_action).

#### Actions that return a collection

An action (on a service or a domain entity object) may return any of the [recognised collection types](#recognised_collection_types). A collection returned by an action will automatically be presented to the user in paged form. The default page size is 20 objects, but this may be overridden for an individual action using the the [PageSize attribute](#_PageSize). If the number of objects in the returned collection is less than a single page full, or if the PageSize has been explicitly set to zero, then the page navigation controls will be disabled.

However, if you know that the collection being returned is likely to contain sufficient objects that paging will be shown, then it is strongly recommend that you return the collection as an IQueryable<T>. With the latter, the objects will automatically be retrieved from the database one page at a time. By contrast, if you return an ICollection<T> the entire collection of objects will be retrieved from the database, but only one page displayed; and this will be repeated for each page request.

**Tip**: The advantage of using IQueryable<T> is so great that it is recommended that you use this by default return for any action returning a collection, unless you have good reason not to.

### Menus

Actions are typically rendered within Menus (unless, for example, you have written a custom view that renders one or more actions as free-standing buttons). There are two principle kinds of menu: Main Menus, which provide the starting points for user interactions, and Object Menus which offer actions provided by an object instance. The programming model is easier to understand by starting with object menus.

#### Object Menus

By default, the action menu for a domain object will be constructed from all the actions defined on that object, plus [contributed actions](#contributed_action) (if any). The ordering of actions within the menu will honour any [MemberOrder attributes](#memberorder_attribute). This may be overridden by providing a static Menu method as in the following example:

public static void Menu(IMenu menu) {…}

This method is called during the start-up process. The Naked Objects framework will call it with a framework-generated implementation of IMenu, and which has its Type property set to the type of the object on which the Menu method is called. IMenu provides a number of methods for adding actions to the menu in an explicit order, and creating sub-menus. For example:

public static void Menu(IMenu menu) {

menu.AddAction("Action2");

menu.AddAction("Action1");

var sub1 =menu.CreateSubMenu("Sub1");

sub1.AddAction("Action3");

var sub2 = menu.CreateSubMenu ("Sub2");

sub2.AddAction("Action4");

menu.AddRemainingNativeActions();

menu.AddContributedActions();

}

In this method, the ordering of actions is specified explicitly. Also sub-menus are defined. IMenu defines a number of other helper methods, not shown above, for constructing the custom menu.

Note that the action names are specified in the same format as the method names in code - even if any of the actions have been given a customised name using the [Named](#_Named) or [DisplayName](#_DisplayName) attributes.

**Tip:** C# 6 provides a nameof keyword and we recommend using this if possible, since it handles the renaming of actions automatically. e.g:

menu.AddAction(nameof(Action2));

*Sub-menu names should not contain underscores* (e.g. "My\_sub\_menu" ) – as this would cause them to be rendered incorrectly.

#### Main Menus

Main menus offer actions that are defined on services. The menus are specified via the MainMenus method on NakedObjectsRunSettings. The following shows an example:

public static IMenu[] MainMenus(IMenuFactory factory) {

var customerMenu = factory.NewMenu<CustomerRepository>();

CustomerRepository.Menu(customerMenu);

return new IMenu[] {

customerMenu,

factory.NewMenu<OrderRepository>(true),

factory.NewMenu<ProductRepository>(true)

};

The framework calls this method on start-up, passing in an implementation of IMenuFactory, which provides various methods for creating menus. In the above example, a default menu is created for each of two services (OrderRepository and ProductRepository), the true parameter indicating that *all* the actions from that type should be added into the menu, recognising the order implied by the MemberOrder attributes on those actions (if specified). The CustomerRepository has its own static method (Menu) that defines a fully custom menu, which looks like this:

public static void Menu(IMenu menu) {

menu.AddAction("FindCustomerByAccountNumber");

menu.CreateSubMenu ("Stores")

.AddAction("FindStoreByName")

.AddAction("CreateNewStoreCustomer")

.AddAction("RandomStore");

menu.CreateSubMenu("Individuals")

.AddAction("FindIndividualCustomerByName")

.AddAction("CreateNewIndividualCustomer")

.AddAction("RandomIndividual");

menu.AddAction("CustomerDashboard");

}

This method has the same structure and capabilities as the optional static Menu method on a domain object that defines an [object menu](#_Object_Menus). However, a static Menu method on a *service* will  *not* be called automatically – and must be called explicitly from the NakedObjectsRunSettings.MainMenus method, with a suitable menu object obtained from the factory, as shown above. This difference is deliberate, because Main Menus need not have a 1:1 relationship to underlying services.

In the example above, the three main menus still *do* correspond to three different services, but this is not a requirement. A main menu may combine actions from different services and, conversely, the actions on a single service might be split across multiple main menus. There need be no correspondence at all. When adding actions from multiple services, you will need to set the Type property of the IMenu to the service on which the action exists *before*  calling AddAction, otherwise you will get an exception indicating that the requested action does not exist. For example:

public static void Menu(IMenu menu) {  
 menu.WithName("My Hybrid Menu");  
 menu.Type = typeof(ServiceA);

.AddAction("ActionA1")

.AddAction("ActionA2")

menu.Type = typeof(ServiceB); // changing the type for the next additions

.AddAction("ActionB1")

.AddAction("ActionB2")  
 .CreateSubMenu(("Cs").  
 .Type = typeof(ServiceC) // changing the type for the sub-menu now  
 .AddAction("ActionC1")  
 menu.AddAction("ActionB3") // menu itself still has type ServiceB

}

### Recognised method

Naked Objects uses 'programming by convention': methods with specific names are recognised by the framework as intending to specify a certain behaviour. Some of these recognised methods are standalone (e.g. Title(), Created()), but the majority take the form of method prefixes (e.g. Validate<Something>, Choices<Something>) which provide behaviour in relation to either a specific property or action. See the complete list of [Recognised Methods](#_Recognised_Methods).

### Recognised attribute

Naked Objects recognises a set of attributes that may optionally be applied to domain objects, properties, actions, or action parameters, to alter presentation or behaviour. For the full list of recognised attributes see [Recognised .NET attributes](#_Recognised_.NET_attributes) and [NakedObjects.Attributes](#_NakedObjects.Attributes).

### View Model

Naked Objects is designed to expose persistent domain objects (or 'entities') directly to the user. This pattern encourages the development of a domain model that corresponds directly to the user's own representation of the problem domain.

On occasions, it may be desirable to present objects to the user that do not correspond directly to a persistent domain object - perhaps amalgamating information and/or functional behaviours from more than one underlying domain object. In these circumstances, you can use a View Model.

#### IViewModel, IViewModelEdit, and IViewModelSwitchable

In Naked Objects, a View Model is coded the same way as a domain object but must implement one of these three interfaces: IViewModel, IViewModelEdit, or IViewModelSwitchable. All three of these interfaces require that the following two methods are implemented:

* DeriveKeys must return a string array that defines the key(s) to this object. Typically these will be derived from the key(s) of the persistent object(s) that this view model represents, but could also directly represent value properties.
* PopulateUsingKeys takes the same array of keys as a parameter and is used to re-populate the view model whenever it is referenced in a request (either to refresh the view or to invoke an action on the view model, for example).

These two methods are needed in order that the Naked Objects framework running on the server - which adopts a stateless pattern - can re-create the instance of the view model each time the client makes a request of it, making it appear (to the client) as if it were a persistent object.

Implementation of those two methods is best illustrated by example. In the code below, CustomerComparison holds a reference two separate Customer objects, for the purposes of comparison. Other properties (not shown in the code below) derive information from these two customers (or from associated objects) for presentation in the view model.

[NotMapped] //Needed if working with EF Code First, so that no table is created

public class CustomerComparison : IViewModel

{

public IDomainObjectContainer Container { set; protected get; } //Injected service

public virtual Customer Customer1 { get; set; }

public virtual Customer Customer2 { get; set; }

public string[] DeriveKeys()

{

return new string[] { Customer1.Id.ToString(), Customer2.Id.ToString() };

}

public void PopulateUsingKeys(string[] keys)

{

int id1 = int.Parse(keys.ElementAt(0));

Customer1 = Container.Instances<Customer>().Single(x => x.Id == id1);

int id2 = int.Parse(keys.ElementAt(1));

Customer2 = Container.Instances<Customer>().Single(x => x.Id == id2);

}

//Properties (not shown), derive their information from the two persistent Customers.

}

Note: It is not necessary to have a reference to the Customer object, if you don't want it. The view model could simply hold the Customer's Id.

#### Instantiating a view model

To use any View Model, it should be created by means of a special method on the domain object container: NewViewModel, as shown in the example below:

var cc = Container.NewViewModel<CustomerComparison>();

cc.Customer1 = customer1;

cc.Customer2 = customer2;

return cc;

If your View Model is designed to present details from a single persistent domain object (optionally including information from other objects that are associated with that single object) - which is the most common scenario - then you may simply extend the NakedObjects.ViewModel<T> helper class. For example:

[NotMapped] //Needed if working with EF in Code First mode, so no table is created

public class MyCustomerViewModel : ViewModel<Customer>

{

//Properties (not shown), derive their information from the 'Root' property (defined on ViewModel), which will be of type Customer, for example:

public string Name {

get {

return Root.Name;

}

}

}

#### Distinctions between the three types of view model

* Objects that implement the IViewModel interface. To the client, these are indistinguishable from persistent objects. However their properties are never modifiable directly. They may have actions, and those actions may have parameters.
* Objects that implement the IViewModelEdit interface. These are informally referred to as ‘forms’ – and are intended for inputting information. Their properties are typically modifiable (though they may include unmodifiable ones also). They may include actions, *but these actions cannot take parameters*. You might choose to write an action called Save (or Next or Cancel) on your editable view model, but it is up to you to decide what that does (for example it might use the entered values to create one or more persistent objects).
* Objects that implement the IViewModelSwitchable interface. This requires implementation of the IsEditView() method in the domain code. When this returns true, then the object behaves exactly like one implementing the IViewModelEdit interface. When the method returns false, it behaves like an object implementing IViewModel.

#### Concurrency Checking and View Models

For all uneditable view models, and even for some editable ones, concurrency checking is irrelevant. Where concurrency is relevant - for example where an editable view model is being used to update an underlying persistent entity - then the Naked Objects concurrency checking mechanism can be applied, in much the same way as for a persistent entity. You will need to have a single property on the view model, marked up with [ConcurrencyCheck] that acts as the version (it can be hidden from the user). In a view model it would be common to make this version property derive from an equivalent version property on the underlying persistent entity - so that if that entity has been updated by another user then the submission of new values will fail. If your view model represents more than one underlying persistent entity, and you want to guard against updates to any of them, then you can make your derived version property be a hash of all the underlying ones.

### Service

Services perform three roles in a Naked Objects application:

* To provide methods for creating and retrieving domain objects where the user does not have an existing object to navigate from. Services that perform this role are often referred to as [Factories & Repositories](#_Repository).
* To provide a bridge to external functionality. See [External or System service](#_External_or_System).
* To provide functionality that is to be shared by multiple classes of domain objects which do not necessarily have any common superclass. This is achieved through the concept of contributed actions whereby methods are written on a service but appear to the user as actions on a domain object. See [Contributed Action](#_Contributed_action).

In whichever of these roles, a service is just an ordinary POCO class but without any state - just methods. Just like a domain object it does not have to inherit from any special class, nor implement any interface, nor include any specific attribute. What makes it a service is simply that it is [registered as a service](#_Registering_domain_services). Registering the service instructs the NOF to inject that service into any domain object that needs access to it. See [Injection of domain services into domain objects](#_Dependency_Injection).

### Factories and Repositories

A repository is a service that provides method(s) for finding one or more domain objects. A factory is a service that provides method(s) for creating a new instance of a domain object class. Some domain modellers feel strongly that these two roles should be kept distinct; others see merit in merging them (in which case the term ‘repository’ is sometimes used to mean repository/factory). Naked Objects permits either style of coding.

Incidentally, there is no formal reason to stick to the idea of having a separate repository for each class: you might have a single repository to deal with a number of closely-related domain classes. And there is no need for any repository methods to deal with classes that are always retrieved by navigating from an associated object - as that functionality is provided automatically by the Naked Objects framework. You only need a repository when it is necessary to create and/or retrieve objects directly, rather than by navigating from an associated object.

To write a repository you simply need to provide a write-only property into which the framework can inject an [IDomainObjectContainer](#_The_Domain_Object_1), and then make use of the NewTransientInstance and Instances methods on that container, as illustrated in the following example:

public class CustomerRepository {

public IDomainObjectContainer Container { set; protected get; }

public Customer CreateNewCustomer() {

return Container.NewTransientInstance<Customer>();

}

public Customer FindCustomerByNumber(string num) {

var query = from obj in Container.Instances<Customer>()

where obj.Number == num select obj;

return query.FirstOrDefault;

}

}

#### Accessing one Repository from inside another

Sometimes, it is desirable to be able to access one Repository from within another: for example accessing the ProductRepository from within the OrderRepository. This is handled by [dependency injection](#dependency_injection).

### External or System service

The second role that services perform within a Naked Objects application is as a bridge to external functionality. The following are examples of what we mean by bridging domains:

* Linking to functionality that already exists, or has to exist, outside of the Naked Objects application, such as pre-existing services, or functionality within legacy systems.
* Bridging between technical domains, such as between the object domain and the relational database domain, or the email domain.

The following example shows a service to send an email message. The service is defined by an interface IEmailSender, which in turn defines as single method SendTextEmail. The following code defines a particular implementation of that service interface that uses the SmtpClient to send the message.

public class SmtpMailSender : AbstractService, IEmailSender

{

private static string SMTP\_HOST\_NAME = "localhost";

private static string SMTP\_USER = "admin@example.com";

public void SendTextEmail(string toEmailAddress, string text)

{

SmtpClient client = new SmtpClient();

client.Host = SMTP\_HOST\_NAME;

MailMessage message = new MailMessage();

message.Sender = new MailAddress(SMTP\_USER);

message.From = new MailAddress(SMTP\_USER);

message.To.Add(new MailAddress(toEmailAddress));

message.Subject = "Expenses notification";

message.Body = text;

client.Send(message);

}

}

### Contributed action

A contributed action is an action that is defined on a service, but which is presented to the user as an action on an individual domain object or a collection of objects.

This is a very powerful feature of Naked Objects, but it is one that takes a bit of getting used to. A contributed action has some conceptual similarity to the .NET programming concept of an 'extension method', but they are not the same. Extension methods may be used within a Naked Objects application to provide internal functionality, but extension methods are not recognised as actions by Naked Objects. (Extension methods are also static, whereas contributed actions are instance methods on a service).

#### Actions contributed to individual objects

If an action defined on any service takes a domain object as one of its parameters, then adding a [ContributedAction](#_ContributedAction) attribute to that parameter, means that action will automatically be contributed to any domain object of that type (or, if the parameter is an interface type, any domain object that implements that interface). The attribute may optionally specify a sub-menu that the action is to be contributed to.

In the following example, the method CreateNewTask is defined on a service called TaskContributedActions. The method will appear as an action on any domain object that implements ITaskContext - under a sub-menu called Tasks.

public class TaskContributedActions : AbstractService

{

public Task CreateNewTask([ContributedAction("Tasks")] ITaskContext context)

{

var task = Container.NewTransientInstance<Task>;

task.Context = context;

return task;

}

//...

}

Because the method CreateNewTask takes just one parameter, the action will appear on the domain object as though it was a zero-parameter action - under the covers the method on the service will be called, but with the domain object from which it was initiated as the parameter. If the method had multiple parameters then it would appear on the domain object as a multi-parameter method which, when invoked, will return a dialog box showing all *other* parameters.

#### Actions contributed to query results

As well as being contributed to individual domain objects, actions may also be contributed to collections of domain objects that are returned as the result of queries.

Any such action needs to be declared on a service, just like other contributed actions, but will take an IQueryable<T> (where T is a domain entity type) as one of the parameters. If the parameter is marked up with the [ContributedAction](#_ContributedAction) attribute this method will show up on the action menu on any IQueryable<T> returned as the result of some kind of finder or query action. You may invoke the action on all the elements in the query result, or you may select a sub-set of objects - in which case, behind the scenes, a new collection will be formed from just those objects selected and this new collection will be passed into the action method as a parameter.

This is best illustrated by example, taken from an accounting application, where the class Transaction represents an accounting transaction and has an instance method MarkAsReconciled. In addition there is a method also called MarkAsReconciled, declared on a service called TransactionContributedActions, but which takes an IQueryable of Transaction as its parameter:

public void MarkAsReconciled(

[ContributedAction()] IQueryable<Transaction> transactions) {

foreach (Transaction t in transactions.ToList())

{

t.MarkAsReconciled();

}

}

**Note**: the sub-menu parameter of the ContributedAction attribute, is ignored for query-result-contributed actions. (It would be unusual to have more than just a handful of actions contributed to any one type of collection.

The following limitations apply to query-result-contributed actions:

* Query-result-contributed actions may not themselves return a collection. However, if you have a requirement to return a collection of objects from the action, then you can do this by returning a View Model that is capable of re-generating the same collection on demand.
* Actions may be contributed only to IQueryable<T> and not to, say, ICollection<T> or IEnumerable<T>.
* Where the results of a query are large enough to be presented as more than one page of results, then selection of objects (for the purpose of invoking a contributed action to that selection) operates only within a single page. It is not possible to make a selection that spans multiple pages of a query result. However, it is possible to expand the [page size](#_PageSize).

### Injection of domain services into domain objects

Naked Objects has in-built mechanism to inject registered domain services into domain entities (or, indeed, into other domain services). This mechanism is separate from the Dependency Injection mechanism (Unity, by default) that manages the [configuration of the system](#_Application_configuration).

To use this capability, simply provide a property with a protected get and with the type of service required. This type may be a concrete class (such as CustomerRepository) or it may be an interface (such as ICustomerRepository). The advantage of the latter is that you can then have multiple implementations of that interface. It also means that you can have a single class that implements several separate services each defined by a separate interface (this is sometimes useful during prototyping in particular).

This is illustrated in the following example, where a Product object has a method to find products of similar colour, which it delegates to the ProductRepository.

public class Product {

public string Colour { }

public ProductRepository ProductRepository { set; protected get; }

public IQueryable<Product> FindOtherProductsOfSameColour() {

return ProductRepository.ListProductsForColour(this.Colour);

}

}

Tip: Use the injs snippet to create this code.

It is also possible to register multiple services that implement a common interface, and inject them as an array, as illustrated below:

public class Product {

public IPricingMechanism[] PricingMechanisms { set; protected get; }

public void CalculateBestPrice() {

this.Price = PricingMechanisms.Min(pm => pm.Price(this);

}

}

Note however that if you do have multiple implementations then you *must* use an array. A property allowing only a single implementation, i.e:

public IPricingMechanism PricingMechanism { set; protected get; }

will result in a run-time exception being thrown. However, the reverse is fine: you can inject a single implementation into a matching array property.

### The Domain Object Container

The design ethos of Naked Objects is that it is the responsibility of the framework to determine how the application should run, by calling upon the domain objects - not vice versa. However, there are a few situations where it is necessary for the domain objects to be able to communicate with the framework.

This is done via the domain object container, which is defined by the interface [NakedObjects.IDomainObjectContainer](#_NakedObjects).

The container is accessed by means of dependency injection. Any domain object (whether it represents an entity or a service) that needs to call any of the container methods, simply needs to provide a property of type IDomainObjectContainer. The property should have a protected get method.

public IDomainObjectContainer Container { set; protected get; }

The framework recognises that this property is not intended for display to the user, so there is no need to mark it as Hidden. The property is not persisted.

Tip: Use the Injected Container snippet (shortcut injc) for creating this property.

The object may then invoke any of the methods on the container e.g.:

var cust = Container.NewTransientInstance<Customer>();

#### IDomainObjectContainer methods

NakedObjects.IDomainObjectContainer (which may be found in NakedObjects.Attributes.dll) defines the members available on the container object. An implementation of this interface will be automatically injected by the container into any object that provides a property of type IDomainObjectContainer as described above.

* NewTransientInstance<T> returns a new object of a specified type in a transient (not yet persistent) state. If this transient object is returned to the user then it will appear in edit mode ready to be completed and then saved with the Save button. Alternatively, the programmer may persist the transient object with ...
* Persist(ref object). Note that the object must be passed into the method ‘by reference’. This is because the object will be replaced by a new object, with the same state, but now managed by the Entity Framework. An object is only ever persisted once - thereafter it remains in a persistent state (and is managed by the object Persistor) and its properties are updated as needed. Calling Persist on an object that is already persistent will throw an error.
* IsPersistent(object) allows you to check if a particular object is already persistent.
* DisposeInstance(object)allows you to delete a persistent object. The programmer must take responsibility to ensure that any references to this object from other associated objects are cleared as part of the same transaction.
* Instances returns all the instances of a specified type as an IQueryable, upon which LINQ queries can then be performed. There are two overloaded versions of this method: one is templated and should be used wherever the type of object being retrieved is known statically. The other takes a System.Type as a parameter and may be used where the type is not known statically.
* InformUser(string) sends an informational message to the user.
* WarnUser(string) sends a warning message to the user.
* RaiseError(string) sends an error message to the user.
* Resolve(object) ensures that an object has been fully resolved from the Persistor. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework. **Note:**  Resolve and ObjectChanged (below) are largely a hangover from older versions of Naked Objects and might be removed from the interface in a future release.
* ObjectChanged(object) is used to notify Naked Objects that an object has changed and that those changes may need to be notified to the object Persistor and/or to the user interface. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework.
* Principal (property) returns a System.Security.Principal.IPrincipal, from which you may obtain the user's identity and thence the name.
* AbortCurrentTransaction. Any uncaught exception thrown within an action method will automatically abort the transaction. If you wish to abort a transaction without throwing an exception you may use this method.
* NewTitleBuilder() (plus two overloads). Returns an implementation of ITitleBuilder, which provides convenience methods for building object titles from properties and references.
* TitleOf(object) As an alternative to using NewTitleBuilder, this method provides a convenient way to obtain the title of another domain object, without having to know how that title is defined.
* Refresh(object). Ensures that the object is updated with the latest values from the object store.
* NewViewModel<T>. Broadly equivalent to NewTransientInstance, but for View Models (which are never persisted).
* GetService<T>. Mechanism to obtain a specific service othe then by [dependency injection](#dependency_injection).

## Application configuration

As an application developer, there are things that you need to specify in order to run an application under the Naked Objects framework. This is done via specific methods on the NakedObjectsRunSettings class – which is found your run project(s). These methods are described in the following sections.

### Specifying the Namespace(s) that cover your domain model

At start-up time, the Naked Objects ‘Reflector’ introspects the domain model to build an internal metamodel. It does this by starting from the domain services that you have registered (see below) and, from the method signatures on those services, traversing the whole model. It is necessary to specify the namespace(s) that encompass your model, so that the reflector does not disappear off into, for example, library code that provides only technical capabilities, not domain objects. You specify the namespaces using the ModelNamespaces property on NakedObjectsRunSettings:

private static string[] ModelNamespaces {

get {

return new string[] {"Model1","Model2"};

}

}

If all your domain classes (including services) fall within a single root namespace, you need register only that root. Note that the namespace(s) need only cover the domain classes and services, not the run project classes, which do not need to be introspected.

### Registering domain services

All [domain services](#service), whether they provide methods intended to be menu actions, or just technical capabilities, and that may need to be [injected](#_Injection_of_domain) into domain objects (or into other domain services) should be registered in the services property on NakedObjectsRunSettings, for example:

private static Type[] Services {

get {

return new Type[] {

typeof(CustomerRepository),

typeof(OrderRepository),

typeof(OrderContributedActions),

typeof(CustomerContributedActions)

};

}

}

### Specifying any types that will not ordinarily be discovered by the reflector

For most simple domain models, the framework’s introspector will be able to find all the domain types just by traversing the code from the methods on the registered services. However, in more complex domain models, you might have domain types that cannot be reached this way. The most common example of this is where a domain type has sub-types that are created/retrieved inside a method, but which are never explicitly used in a method signature.

If an unknown domain type is encountered at run-time an error will be thrown because the framework will not be able to find any metadata about it. The exception message will typically be a NakedObjectsSystemException with the message ‘*failed to find spec for [type]’*.

To avoid this, you need to register any of these undiscoverable domain types in the Types property on NakedObjectsRunSettings, for example:

private static Type[] Types {

get {

return new[] {

typeof (CustomerCollectionViewModel),

typeof (OrderLine),

typeof (QuickOrderForm)

};

}

}

It is not necessary to register all domain types - only the undiscoverable ones. However, there is no harm in explicitly registering a type that may also be discovered automatically. Indeed, if your model contains only domain types, you might be able register them all in one go using something like this code:

private static Type[] Types {

get {

var allTypes = AppDomain.CurrentDomain.GetAssemblies()  
 .Single(a => a.GetName().Name == "AdventureWorksModel").GetTypes();

return allTypes.Where(t => (t.BaseType == typeof(AWDomainObject)) &&   
 !t.IsAbstract).ToArray();  
 }  
}

### Configuring the EntityObjectStore

The EntityObjectStore is configured to operate with one or more DbContexts as follows:

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {

var config = new EntityObjectStoreConfiguration();

config.UsingCodeFirstContext(() => new MyDbContext());

return config;

}

**Note**: as of version 7.1, Naked Objects now only works using Entity Framework’s ‘code first’ pattern - it no longer supports the older pattern that used .edmx files.

Other options may also be specified on the configuration. This includes the ability to register types that are not persisted. The advantage of doing this is that Naked Objects will not both to interrogate the DbContext(s) to see if that type is known about.

### Configuring Authorization

(Read the section on [Authorization](#_Authorization) first).

You will need create an instance of an implementation of IAuthorizationConfiguration. A convenient place to create this instance is in NakedObjectsRunSettings, where there is already a default (null) implementation. For example:

public static IAuthorizationConfiguration AuthorizationConfig() {

var config = new AuthorizationConfiguration<MyDefaultAuthorizer>();

config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");

config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");

config.AddTypeAuthorizer<Bar, MyBarAuthorizer>();

return config;

}

Each authorizer must implement NakedObjects.Security.INamespaceAuthorizer or ITypeAuthorizer<T> where T is a specific domain type. The ‘default authorizer’ (and only that one) must implement ITypeAuthorizer<object>

This authorization config must then be registered in Unity.config, and you must also register the AuthorizationManager as a facet decorator. If you created your authorization config in NakedObjectsRunSettings then the registrations will happen automatically, because of this default code:

if (NakedObjectsRunSettings.AuthorizationConfig() != null) {

container.RegisterType(typeof(IFacetDecorator), typeof(AuthorizationManager),   
 "AuthorizationManager", new ContainerControlledLifetimeManager());

container.RegisterInstance(typeof(IAuthorizationConfiguration),   
 NakedObjectsRunSettings.AuthorizationConfig(), new ContainerControlledLifetimeManager());  
}

### Configuring Auditing

(Read the section on [Auditing](#wif_authorization) first).

You will need to create an instance of an implementation of IAuditConfig. A convenient place to create this instance is in NakedObjectsRunSettings, where there is already a default (null) implementation. For example:

private static IAuditConfiguration MyAuditConfig() {

var config = new AuditConfiguration<MyDefaultAuditor>();

config.AddNamespaceAuditor<MyAuditor1>("MySpace.Foo");

config.AddNamespaceAuditor<MyAuditor2>("MySpace.Bar");

return config;

}

You will need to have a default auditor, and then you may optionally register an unlimited number of other auditors for specific namespaces each of which must implement NakedObjects.Audit.IAuditor.

This audit config must then be registered in Unity.config, and you must also register the AuditManager as a facet decorator. If you created your Authorization config in NakedObjectsRunSettings then the registrations will happen automatically, because of this default code:

if (NakedObjectsRunSettings.AuditConfig() != null) {

container.RegisterType(typeof(IFacetDecorator), typeof(AuditManager),   
 "AuditManager", new ContainerControlledLifetimeManager());

container.RegisterInstance(typeof(IAuditConfiguration), NakedObjectsRunSettings.AuditConfig(),  
 new ContainerControlledLifetimeManager());  
}

### Configuring Profiling

(Read the section on [Profiling](#_Profiling) first).

You need to configure your implementation of IProfiler, together with the set of events to be profiled, using an IProfileConfiguration. A convenient place to do this is in NakedObjectsRunSettings using the concrete ProfileConfiguration<T> class:

public static IProfileConfiguration ProfileConfig() {

var events = new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation }; //etc

return new ProfileConfiguration<MyProfiler>() { EventsToProfile = events };

}

This profile configuration must then be registered in the UnityConfig file. ProfileManager must also set up as a facet decorator, for example:

if (NakedObjectsRunSettings.ProfileConfig() != null) {

container.RegisterType(typeof(IFacetDecorator), typeof(ProfileManager),   
 "ProfileManager", new ContainerControlledLifetimeManager());

container.RegisterInstance(typeof(IProfileConfiguration),   
 NakedObjectsRunSettings.ProfileConfig(),   
 new ContainerControlledLifetimeManager());

}

### Configuring the RestRoot

The root URL for the Restful Objects API is specified on the RestRoot property of the RestfulObjectsConfig class, for example:

public static string RestRoot {

get { return "rest"; }

}

## System configuration using the Unity framework

Naked Objects uses ‘dependency injection’ (DI) to wire-together the various components of the framework. Naked Objects uses the Microsoft Unity framework to manage this DI. The App\_Start folder of your run project contains a file UnityConfig.cs. This is where the application is configured. The RegisterTypes method defines all the different components that are used within the framework.

It is not necessary for an application developer modify any of the code in the RegisterTypes method in order to run an application.

One example of where you might need to edit the UnityConfig.cs file is to …

#### Configure Localization and Internationalization

See [Internationalisation / Localisation](#_Internationalisation_/_Localisation). To use internationalisation, you need to register the I18NManager as a ‘Facet Decorator’ with the following line of code:

container.RegisterType(typeof(IFacetDecorator), typeof(I18NManager), "II18NManager", new ContainerControlledLifetimeManager());

## The Naked Objects programming model

The Naked Objects programming model defines the ways in which domain model code is recognised by the Naked Objects framework. Broadly there are two main aspects to this. The first is a set of recognised conventions - which include a set of .NET types, .NET attributes, and method names that will be interpreted by the Naked Objects framework when you run your domain model against it.

The second is a set of specific artifacts that are installed via the NakedObjects.ProgrammingModel NuGet package, in three assemblies: NakedObjects.Attributes, NakedObjects.Types, and NakedObjects.Helpers, and which you may choose to reference within domain model code.

### Recognised Value Types

Naked Objects recognises the following .NET types as 'value objects':

* System.Boolean
* System.Byte
* System.SByte
* System.Byte[] (represents a 'blob' - typically an attached file)
* System.Char
* System.Decimal
* System.Double
* System.Enum (strictly speaking, sub-classes of Enum that are defined and used within the domain model)
* System.Float
* System.Single
* System.Int16
* System.Int32
* System.Int64
* System.SByte
* System.UInt16
* System.UInt32
* System.UInt64
* System.String
* System.DateTime
* System.TimeSpan
* System.Guid
* System.Drawing.Color

### Recognised Collection types

Collections are recognised by Naked Objects in two different contexts:

1. As returned by an action (on an object or service), for example an action that finds,or even creates, multiple objects
2. As a property on on object, representing a multiple association

These are defined below.

#### Collections returned by an action

In the this context Naked Objects recognises as a collection any implementation of IEnumerable<T> where T is a domain entity type (class or interface). For example, it will recognise any of the following:

public IEnumerable<Customer> Xxx() {}

public IQueryable<IPerson> Xxx() {}

public ICollection<IDocument> Xxx() {}

public IList<Employee> Xxx() {}

public SalesOrder[] Xxx() {}

It will also recognise an untyped collection (System.Collections.ICollection), though it is recommended that you always type the collection whenever possible as this gives the option to render the results in table form.

The framework deliberately does not recognise as collections implementations of IEnumerable<T> *where* T *is a value type, or any other type not recognised as a domain entity type*. For example, it will not recognise any of the following:

public IEnumerable<decimal> Xxx() {}

public IQueryable<DateTime> Xxx() {}

public ICollection<int> Xxx() {}

public IList<IComparable> Xxx() {}

public String[] Xxx() {}

#### Collections as properties on an object

For a property that represents a multiple association, Naked Objects will recognise any implementation of ICollection<T>, for example:

public ICollection<IDocument> Xxx {get{} set{}}

public IList<Employee> Xxx {get{} set{}}

public SalesOrder[] Xxx {get{} set{}}

Properties returning IQueryable<T> or IEnumerable<T> are not recognised by Naked Objects.

### Recognised .NET attributes

Naked Objects recognises the following .NET attributes.

#### ConcurrencyCheck

System.ComponentModel.DataAnnotations.ConcurrencyCheck

Allows you to specify the property on an object that participates in concurrency checking. This could be any type of property, but is most commonly a DateTime. You must ensure that the value of this property changes with each update of the object. The typical way to do this is to implement the behaviour in the database; however you may choose to implement the behaviour in your domain model, for example within the Updating life-cycle method.

#### ComplexType

The System.ComponentModel.DataAnnotations.Schema.ComplexType attribute is used to indicate that a domain class is treated as a complex type by Entity Framework (you will need to add the Entity Framework NuGet package to your model project if your want to use it). The annotation is also detected, independently, by the Naked Objects framework: this is why (from NOF 7.0 onwards) *you must annotate all complex types with this attribute, even if you have specified that a type is a complex type using the Entity Framework code-first fluent mappings.*

This means that the annotated type will always stored in-line with its parent object. In the following example, the Person object has a property of type Name, which in turn has properties FirstName and LastName and has been annotated with the ComplexType attribute. This indicates that the FirstName and LastName properties will in fact be stored as though they were direct properties of the Person object - in the same table if you are working with a relational database. It also means that the identity of the Name object is tied to the identity of the Person object - so no other persistent object should attempt to hold a reference directly to that Name.

public class Person {

public virtual Name FullName { get; set; }

}

[ComplexType]

public class Name {

public virtual string FirstName { get; set; }

public virtual string LastName { get; set; }

}

Note that the NOF8 user interface will not allow the properties of a complex type (e.g. the FirstName and LastName properties in the example code above) to be edited directly by the user. If these values need to be updated by the user, the domain programmer will need to provide actions on the root class (Person, above) for updating those properties on the Name.

(See also the [Root](#_Root) attribute).

#### DataType

Adding [DataType(DataType.Password)]to a string parameter on an action, will cause the content to be rendered as an obscured field (e.g. \*\*\*\*\*\*).

This should not be used on string properties as - though it will obscure the content in edit mode - it will not obscure the content in view mode. (Normally, if a password needs to be stored on an object then it would be marked as NakedObjectsIgnore anyway, and should only be update-able through an action method.)

#### DefaultValue

System.ComponentModel.DefaultValue

Allows you to specify, declaratively, the default value for any value input field – meaning an editable property or action parameter. (This is an alternative to using a [default method](#_Complementary_methods)). For example:

public class Invoice

{

[DefaultValue(30)]

public virtual int CreditDays { get; set; }

}

#### Description

System.ComponentModel.Description

May be used as an alternative to the [NakedObjects.DescribedAs](#_DescribedAs) attribute.

#### DisplayName

System.ComponentModel.DisplayName

May be used as an alternative to the [NakedObjects.Named](#_Named_1) atttribute.

#### MaxLength

System.ComponentModel.DataAnnotations.MaxLength

This is an alternative to the [StringLength](#_StringLength) attribute.

The MaxLength attribute indicates the maximum number of characters that the user may enter into a String property, or a String parameter in an action. (It is ignored if applied to a property or parameter of any other type.) For example:

public class Customer

{

[MaxLength(30)]

public virtual string FirstName { get; set; }

}

#### MinLength

System.ComponentModel.DataAnnotations.MinLength

This may be used in conjunction with an AutoComplete method to specify the number of characters that must be typed into the field before a server call will be made to find potential matches. See [How to specify auto-complete for a property](#_How_to_specify) and [How to specify auto-complete for a parameter](#auto-complete_param).

#### Range

System.ComponentModel.DataAnnotations.Range

Range may be used to specify the minimum and/or maximum value for a user input to a numeric or date, property or parameter.

When applied to a date property or parameter, the range values represent the number of days relative to today. Thus Range(1,30) means any day from tomorrow to 30 days from now (inclusive) and Range(-30, 0) means any of the last 30 days including today.

#### RegularExpression

System.ComponentModel.DataAnnotations.RegularExpression

This may be used as an alternative to the [RegEx](#_RegEx) attribute.

#### ScaffoldColumn

System.ComponentModel.DataAnnotations.ScaffoldColumn

This may be used to hide properties in the viewer. It is an alternative to the Hidden attribute. ScaffoldColumn(False) is equivalent to Hidden(WhenTo.Always), and ScaffoldColumn(True) is equivalent to Hidden(WhenTo.Never).

#### StringLength

System.ComponentModel.DataAnnotations.StringLength

Allows you to specify the maximum length that the user may input to a string property or action parameter. This is an alternative to the MaxLength attribute.

**Note**: Naked Objects does not support the MinimumLength property of the StringLength attribute.

### Recognised Methods

This sections defines the explicit method conventions that are defined by the 'Naked Objects Programming Model'. All such methods must be public and should start with an upper-case letter.

Public methods are, by default, are deemed to be action methods that we expect the user to invoke via the user interface:

public void <actionName>([<parameter type> param]...)

public <return type> <actionName>([<parameter type> param]...)

The exception to this rule are methods that follow specific conventions. These may be divided into three broad categories:

* Complementary methods.
* LifeCycle methods
* Other recognised methods.

The specific recognised methods names are listed below under those three headings.

#### Complementary methods

These methods complement an action or a property. They take the form of a prefix followed by the corresponding action or property name. Examples or these prefixes include: Validate, Modify.

A complementary method must be declared on the same class as the action or property that it complements. However, a complementary method may be overridden in a sub-class without having to override the action or property that it complements.

* AutoComplete: Used to generate a dynamic drop-down list of matching objects or values. See [How to specify auto-complete for a property](#_How_to_specify) and [How to specify auto-complete for a parameter](#auto-complete_param).
* Choices: A complementary method used in conjunction with a property or an action. A Choices method specifies a set of explicit choices from which the user must select for a particular property (when editing an object) or for a parameter within an action dialog. See[How to specify a set of choices for a property](#property_choices) and [How to specify a set of choices for a parameter](#action_choices) .
* Default: A complementary method used in conjunction with a property or an action. See [How to specify a default value for a property](#property_default) and [How to specify a default value for a parameter](#action_defaults).
* Disable: A complementary method used in conjunction with a property or an action. The Disable dynamically controls whether a field is editable, or an action can be initiated. If a String is returned the field or action is disabled and the string is made visible to user to inform them why it is disabled. If the method returns null then the field or action remains enabled. See [How to prevent the user from modifying a property](#_How_to_prevent) and [Disabling an action based on the state of the object](#_Disabling_an_action).
* Hide: A complementary method used in conjunction with a property or an action. The Hide method allows a property or action to be dynamically hidden from the user, such as hiding a field once it is set up. Returning true makes the property or action invisible. See [How to hide a property from the user](#_How_to_hide) and [How to hide actions](#action_hide).
* Modify: A complementary method used in conjunction with a property. The Modify method is called when the user (rather than the framework) sets a reference or value field. This is typically used to trigger other behaviours such as updating a total. See [How to trigger other behaviour when a property is changed](#_How_to_trigger).
* Validate: A complementary method used in conjunction with a property or an action to validate value(s) entered by the user. See[How to validate user input to a property](#_How_to_validate) and [How to validate parameter values](#action_validate).

#### LifeCycle methods

The following is a list of methods that correspond to various events in the life-cycle of a domain object. If a domain object implements any of these methods (they are all optional) then the framework will call that method whenever the corresponding event occurs.

* Created: Life cycle method called by framework when an object is first created. This is the instance's logical creation. This method will not be called when the object is retrieved from persistent storage into memory.
* Deleted: Life cycle method called by framework when an object has just been removed from the persistent store. At this point the object will exist in memory, but no longer exist in the persistent store.
* Deleting: Life cycle method called by framework when an object is just about to be removed from the persistent store. At this point the object still exists in the persistent store.
* Loaded: Life cycle method called by framework when an object has just been loaded into memory from the persistent store. At this point the object has had its state fully restored. Loaded will be called after the object has been loaded and before the transaction has completed. When retrieving an object via the user interface this means that Loaded will have been called by the time the object appears on the screen. However, if you are processing objects programmatically - whether from within a user action or from an external call - then be aware that the Loaded might not be called on any (or all) of the objects being processed until the very end of the transaction. So if your method involves loading objects and processing them, you cannot assume that Loaded will have been called before you get hold of each object. In general, it is recommended that you use Loaded only for very simple, non-invasive purposes, such as calculating a total for display purposes before an object is returned to the user.
* Loading: Life cycle method called by framework when an object is just about to be loaded from the persistent store. At this point the object exists in memory but has not had its state restored. You should never attempt to reference any non-scalar property within your Loading method. Unlike the other pairs of methods, there may be a considerable time gap between the calling of Loading() and Loaded() on an object - with the latter being called only when the object is first displayed or used programmatically. For most application purposes, Loaded() is a more useful event than Loading().
* OnPersistingError: Life cycle method called by the framework if the object persistor throws an exception when an object is persisted. Typically this will be a DataUpdateException or an OptimisticConcurrencyException. By adding the OnPersistingError method to your code you can intercept this exception and parse its message to establish details. When the OnPersistingError method exits, the framework will still throw an exception to be caught by the user interface. However, you may specify the message for that exception by returning the desired message as a string from your method. There is no point in adding this method to your code unless you want to change the message that is passed to the user - otherwise just leave the exception handling to the framework. Note that if your returned string message contains one or more line-breaks (\n) then any text after the first line-break will be moved into the Details section of the resulting dialog.
* OnUpdatingError: Life cycle method called by the framework if the object persistor throws an exception when an object is being updated. Works in a similar manner to OnPersistingError.
* Persisted: Life cycle method called by framework after a transient object has been persisted. **Important**: unlike Persisting, the Persisted method will be in a separate transaction to the persisting of the object. This is useful because it means, for example, that if the key is database-generated then that generated value will be visible from within the Persisted method, but not in the Persistingmethod. Note, however, that the two transactions will still occur within an over-arching super-transaction, such that any exception occurring within the scope of the Persisted method will cause the whole action to fail i.e. the object will not itself be persisted.
* Persisting: Life cycle method called by framework when a transient object is just about to be persisted via the object store, as part of the same transaction. At this point the object exists only in memory and not in the persistent store.
* Updated: Life cycle method called by framework when a modified persistent object has just been saved to the persistent store. At this point the object in the persistent store will be in its new state.
* Updating: Life cycle method called by framework when a persistent object has just been modified and is about to be saved to the persistent store. At this point the object's data held in the persistent store will not yet have been modified.

#### Other recognised methods

* Title: A Title method, returning a string, is one way to specify the title for an object. See [How to specify a title for an object](#object_title) .
* ToString: If no [title attribute](#_Title), or title method (see above) has been specified, then the framework will call the object's ToString method to get a title for the object.
* Menu. Method to define a menu on an object. See[Object Menus](#_Object_Menus) .
* IconName. This is a recognised method, so will not show up as an action. However, it no longer serves any valid purpose in NOF8 and will be removed in a future release.

### NakedObjects.Attributes

NakedObjects.Attributes.dll includes a set of attributes that are recognised by the framework.

#### ActionOrder

This attribute is now obsolete and will be deleted in a future version.

#### AuthorizeAction

Specifies the users and/or roles to whom an action is to be made available. May be applied to an individual action, or at class level to apply to all actions in that class (including sub-classes). See [Attribute-based Authorization](#_Attribute-based_Authorization).

#### AuthorizeProperty

Specifies the users and/or roles that may view, and, separately, Edit a property. May be applied to an individual property, or at class level to apply to all properties in that class (including sub-classes). See [Attribute-based Authorization](#_Attribute-based_Authorization).

#### Bounded

For immutable objects where there is a bounded set of instances, the Bounded attribute can be used. For example:

[Bounded]

public class County {...}

The number of instances is expected to be small enough that all instances can be held in memory. The NOF8 standard user-interface uses this information to render all the instances of this class available to the user as a drop-down list.

#### ContributedAction

Applied to a specific reference parameter within an action on a service, indicating that that action should be [contributed](#contributed_action) to the action menu on that type of domain object, or, if the parameter is an IQueryable of a domain type - as an [action contributed to a query result](#collection_contributed_actions).

#### DescribedAs

The DescribedAs attribute is used to provide additional information to the user. When used on an action, the text will be rendered (by the standard NOF8 user interface) as a tooltip - visible when the mouse hovers over an action in a menu. When used on a property, or on a specific parameter in an action, the text will be rendered as a ‘placeholder’ (greyed-out text) within the field (but only when the field is empty).

public class Customer {

[DescribedAs("Approximate age (years only)")

public virtual int Age() {get; set;}

[DescribedAs("Create a new order object")

public Order Order() {…}

}

#### Disabled

The Disabled attribute, applied to a property, controls whether or when that property may be modified by the user. For example:

public class Customer {

[Disabled]

public virtual Money InitialCreditRating {...}

}

This attribute can also take a single parameter indicating when it is to be disabled. For example the following code would disable the property until the object has been saved.

public class Customer {

[Disabled(WhenTo.UntilPersisted)]

public virtual Money InitialCreditRating {...}

}

The acceptable values for the parameter are: WhenTo.Always, WhenTo.Never, WhenTo.OncePersisted and WhenTo.UntilPersisted.

#### Eagerly

Eagerly may be applied to a collection, within a class. This indicates that when the object is viewed that collection should be rendered already opened either as a list view, or if there is also a TableView attribute, as a table view.

Note: while the use of Eagerly may be considered advantageous from a user’s viewpoint, developers should be aware that eagerly loading and rendering any collection increases the server loading and response time when viewing an object. Eagerly should therefore be used *sparingly*.

If applied to an action that returns a collection, then the result will be rendered as a table rather than a list. (This should be used in combination with the TableView attribute to specify the columns to be included.)

The syntax for the Eagerly attribute is as follows.

[Eagerly(EaglerlyAttribute.Do.Render)]

public Customer Customer {get; set;}

#### Executed

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### ExcludeFromFindMenu

This attribute is now obsolete and will be deleted from a future version.

#### FinderAction

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### FindMenu

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### Hidden

The Hidden attribute indicates that the member (property, collection or action) to which it is applied should not be visible to the user. It takes a single parameter indicating when it is to be hidden, for example the following code would show the InternalId property until the object has been saved, and then would hide it.

public class Customer {

[Hidden(WhenTo.OncePersisted)]

public virtual int InternalId { get; set; }

}

The acceptable values for the parameter are: WhenTo.Always, WhenTo.Never, WhenTo.OncePersisted and WhenTo.UntilPersisted.

#### IconName

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### Idempotent

Applies to an action, indicating that the action is 'idempotent' - invoking it more than once in succession (with the same arguments) has the same effect as invoking it once.

The effect of this is that in the Restful API (the communication mechanism between the client and the server in Naked Objects), the action will be invoked with the PUT method (rather than the default method of POST) - thereby correctly following the rules of Http.

Note that this is purely an indicator: adding the attribute does not force a method to behave in this fashion. Application developers should therefore take considerable care to ensure that this attribute is applied correctly.

(See also the [QueryOnly](#_QueryOnly) attribute).

#### Immutable

The Immutable attribute may be applied to a class, to indicate that the user may not ever edit any properties on an object. For example:

[Immutable]

public class Country {...}

This attribute can also take a single parameter indicating when it is to become immutable, for example the following code would allow the user to create an email object, specifying its properties before saving, and then prevent any changes by the user once it has been saved.

[Immutable(WhenTo.OncePersisted)]

public class Country {...}

The acceptable values for the parameter are: WhenTo.Always, WhenTo.Never, WhenTo.OncePersisted and WhenTo.UntilPersisted. By default the annotated property or action is always immutable.

#### Mask

The Mask attribute may be used to override the formatting of scalar fields such as dates and numbers. The interpretation of these masks must be configured in the client. See [Configuring Masks](#_Configuring_Masks).

#### MemberOrder

MemberOrder is the recommended mechanism for specifying the order in which fields and/or actions are presented to the user. MemberOrder is specified at the individual member level, on a relative basis. The syntax is:

[MemberOrder(<relative position>)]

where relative position may be a number or a String. The actual sequence is determined by comparing all the values of relative position, using the standard String comparator. If a member does not have a specified order then it will be placed after those that are specified. (Two members may have the same relative position specified, but in such a case the relative ordering of those members will be indeterminate.)

The simplest convention is to use numbers. One option is to use multiples of 10 initially, such that a new member may be added without having to edit existing numbers. An alternative is to adopt the dot-decimal notation, which allows for an indefinite amount of future insertion. For example:

public class Customer {

[MemberOrder(2.1)]

public virtual string Address {get; set;}

[MemberOrder(1.1)]

public virtual string FirstName {get; set;}

[MemberOrder(1.2)]

public virtual string LastName {get; set;}

[MemberOrder(3)]

public virtual Date DateOfBirth {get; set;}

}

This approach is especially useful when dealing with inheritance hierarchies, as it allows sub-classes to specify where their additional members should be placed in relation to those inherited from the super-class.

#### MultiLine

The MultiLine attribute indicates to the user interface that a string property should be rendered over multiple lines.

public class BugReport {

[MultiLine(NumberOfLines = 10)]

public virtual string StepsToReproduce() { get; set; }

}

The NumberOfLines specifies the number of lines to be displayed - if the text contains more lines (subject to any restriction of maximum length imposed by a separate [StringLength attribute](#stringlength_attribute)) then the field will be rendered with vertical scrolling.

(The Width parameter has no effect in NOF 8 and may be obsoleted in a future version).

#### NakedObjectsIgnore

This attribute may be applied to a property or method that never intended to be visible to a user. The difference between marking a public member with NakedObjectsIgnore, rather than Hidden(WhenTo.Always), is that in the former case the Naked Objects Reflector component will not build up any metadata about the member, and nor will it introspect any types used within that member’s signature.

#### Named

The Named attribute is used when you do not want to use the name generated automatically by the system. It can be applied to objects, members (properties, collections, and actions) and to parameters within an action method. See [How to specify a name and/or description for an object](#specify_icon), [Specifying the name for a property](#_Specifying_the_name) and [Hor to specify parameter names and/or descriptions](#_Hor_to_specify).

#### NotContributedAction

This attribute is now obsolete and will be deleted from a future version.

#### NotPersisted

*This attribute is a hangover from earlier versions of Naked Objects and of Entity Framework and we recommend that it not be used on new domain code. It will be obsoleted in a future version. Instead …*

If you wish to create a class that is used for presentational purposes only, and is not persisted, use the [View Model](#view_model) pattern.

If you need to ensure that Entity Framework does not attempt to persist a class, or a specific property, in the database, nor, when creating the database schema build a table for it, use the standard ‘code first’ attribute: System.ComponentModel.DataAnnotations.Schema.NotMapped.

The only remaining situation where NotPersisted may be needed is described [here](#_Adding_further_security) (and this may change in a future version).

#### Optionally

May be applied to a property member on an object, or to a parameter of an action method, to indicate that the value is optional rather than mandatory. See [How to make a property optional (when saving an object)](#_How_to_make).

#### PageSize

This attribute overrides the default page size (of 20 objects) for a specific method that returns a collection of objects. The page size may be set to any integer value.

#### Plural

Where the framework displays a collection of objects it may use the plural form of the object type in view. By default the plural name will be created by adding an 's' to the end of the singular name (whether that is the class name or another name specified using the DisplayName attribute). The framework will also handle words ending in 'y', changing Country to Countries, for example. Where these conventions do not work, the programmer may specify the plural form of the name using the Plural attribute. For example:

[Plural("Children")]

public class Child {...}

#### PresentationHint

PresentationHint is a simple mechanism for providing hints from the domain code to the presentation layer for customising the user interface. The attribute may be applied to a class, a property (or collection), an action or a parameter. The attribute takes a single string (though this might consist of multiple hints separated by spaces).

[PresentationHint("red-background collections-on-right no-tab")]  
public class Foo {

[PresentationHint("rich-text extra-wide")]  
 public string Text {get; set;}

[PresentationHint("button")]  
 public void DoSomething() {}

}

In the Restful API, the hints are added as a custom extension to the relevant representation - as specifically allowed for in the Restful Objects specification. For example:

x-ro-nof-presentationHint: "rich-text extra-wide"

These hints are not used in the standard (Gemini) user interface, but are accessible via the various view models that are used within the templates. Either the generic templates, or specific [custom templates](#_Adding_custom_views), may therefore be easily modified to read and use the presentation hints. For example, you might use this idiom to render the presentation hint (at object level in this example) as an html class, to be picked up by the CSS:

<div class="object view **{{**object.presentationHint**}}**" ng-class="object.color">

#### ProgramPersistableOnly

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### QueryOnly

Applies to an action, indicating that the action is ‘side-effect free’ - invoking it does not change the persistent state of the system.

The effect of this is that in the Restful API (the communication mechanism between the client and the server in Naked Objects), the action will be invoked with the GET method (rather than the default method of POST) - thereby correctly following the rules of Http.

Note that this is purely an indicator: adding the attribute does not force a method to behave in this fashion. Application developers should therefore take considerable care to ensure that this attribute is applied correctly.

(See also the [Idempotent](#_Idempotent) attribute).

#### RegEx

The RegEx attribute may be applied to any property, or to any parameter within an action method, that allows the user to type in text as input. The syntax is:

[RegEx(Validation = <regEx string>)]

(The attribute also has Message and a CaseSensitive properties, but these are not surfaced in the Restful API and so are not available to the user interface.)

The following example shows the RegEx attribute applied to the Email property, to ensure that any entry adheres to the correct form for an email address:

public class Contact {

[RegEx(Validation = @"^[\-\w\.]+@[\-\w\.]+\.[A-Za-z]+$")]

public virtual string Email { get; set; }

}

In the above examples, note the use of the ^ and $ symbols to anchor the text to the start and end of line respectively. This is to ensure that the whole of the input string matches the RegEx. If these symbols are omitted, then the framework will look for any match to the RegEx string within the input text.

#### Root

The Root attribute is intended for use in conjunction with the ComplexType attribute, where an in line object needs to access its parent object programmatically. The in line object should be provided with a property of the same type as the parent object (or any type implemented by the parent) and this property should be marked up with Root. The framework will then inject the parent into that property, in the same way that it might inject a domain object container or other service. The following example builds on an [earlier example](#_ComplexType).

[ComplexType]

public class Name {

[Root]

public virtual Person Person { get; set; }

public virtual string FirstName { get; set; }

public virtual string LastName { get; set; }

}

#### TableView

The TableView attribute allows you to specify the columns that will appear in a table view of a collection; it may be applied to a collection property or to an action that returns a collection, as shown in the example below:

[TableView(true, "Product", "OrderQty", "UnitPrice")]

public virtual ICollection<SalesOrderDetail> Details {...}

[TableView(false, nameof(Task.Description), nameof(Task.DueDate))]

public IQueryable<Task> MyTasks() {...}

The first, boolean, parameter of the attribute determines whether or not the title of each object will be rendered as the first column of the table; it is typically set to false where the title would be effectively duplicated by data in other columns. This parameter is followed by the list of columns that you want rendered, each as a separate string parameter.

Note the following:

* The columns should take the same format as the property name in the code, not as displayed to the user: "OrderQty" not "Order Qty" - the column headers will automatically pick up the correct display format (or the overridden display name if one has been specified). For this reason we strongly recommend using the nameof keyword introduced in C# 6.0, as in the second example above.
* [TableView(true)] will result in a single-column table with just the title of the object, equivalent to a 'list view'.

#### Title

When applied to a property, the Title attribute signifies that property should be used as the title for the object. (This will override a [title method](#_Other_recognised_methods) if one existed). Normally, this should be applied to a property containing a value type such as String, or Date. It may be applied to a reference property, indicating that the title of the object in that reference property should be used as the title of the object that owns the property also. However, this is not a recommended practice as it would force the loading of the reference property earlier than would otherwise be the case - with possible performance implications.

#### TypicalLength

This attribute has no effect from NOF 8 onwards, it will be obsoleted in a future version.

#### ValidateProgrammaticUpdates

If the ValidateProgrammaticUpdates attribute is applied to a class, then all forms of property validation (including Validate... methods, Range, RegularExpression and StringLength attributes, and the enforcement of any mandatory/required properties) will be enforced by the framework even when an object is updated (or persisted for the first time) programmatically. Without this attribute, these forms of validation are enforced only when changes are made via the user interface.

If any of the validation rules are violated then a DomainException will be thrown.

Note that adding ValidateProgrammaticUpdates will not automatically test that a value is consistent with the set of choices offered by Choices method. If the latter check is required, then this should be implemented as a Validate method.

### NakedObjects.Types

The NakedObjects.Types assembly provides a number of interfaces and classes that are explicitly recognised by the Naked Objects framework, and that may be used within a domain model.

#### NakedObjects

* DomainException. All exceptions that are generated within the application domain code should inherit from this class or throw it directly as this allows the Naked Objects Framework to discriminate between potential framework errors, infrastructure errors, and exceptions raised in application code.
* IDomainObjectContainer. See [IDomainObjectContainer methods](#IDomainObjectContainer).
* IViewModel, IViewModelEdit, IViewModelSwitchable. See [View Model](#_View_Model).
* ITitleBuilder. See [How to specify a title for an object](#object_title).

#### NakedObjects.Async

* IAsyncService. Interface representation of the AsyncService class (in NakedObjects.Core.dll) - for injection into domain code. See [How to run multiple threads asynchronously](#_How_to_run).

#### NakedObjects.Audit

* IAuditor. Allows domain programmers to define an auditing service. See [Configuring Auditing](#_Configuring_Auditing).
* INamespaceAuditor. Specific sub-type of IAuditor, where the methods will only be called in relation to types that fall within the specified namespace.

#### NakedObjects.Menu

* IMenu. See [Object Menus](#_Object_Menus).

#### NakedObjects.Profile

* IProfiler. See [Profiling](#xat).
* ProfileEvent an enum defining the various events that may be detected for profiling.

#### NakedObjects.Redirect

* IRedirected. Implemented by a 'stub' class that acts as a proxy for an object managed on another server.
* IRedirectedObject and IRedirectedService. These are obsoleted and will be deleted from a future version.

#### NakedObjects.Security

* INamespaceAuthorizer. An implementation of this interface provides authorization for a single fully-qualified type, or for any types within a namespace. See [Custom Authorization](#_Custom_Authorization).
* ITypeAuthorizer. Implement this interface to manage authorization for a specific class of domain objects rather than a whole namespace.

#### NakedObjects.Snapshot

* IXmlSnapshot. Interface definition of the class IXMLSnapshotService that will be generated by the XMLSnapshotService (see below).
* IXmlSnapshotService. Interface definition for the class XmlSnapshotService (defined in NakedObjects.Snapshot.Xml.dll) allowing this service to be injected into domain code without requiring a reference to the framework.

#### NakedObjects.Value

* FileAttachment. See [How to handle File Attachments](#file_attachment).
* Image provides similar functionality to FileAttachment, but displays the contents as an in-line image rather than as a link. See [How to display an image](#_How_to_display).
* IStreamResource. Interface implemented by both FileAttachment and Image. (Not intended to be used directly within domain code.)

### NakedObjects.Helpers

The NakedObjects.Helpers.dll contains a number of helper classes, extension methods, and other artifacts that you may find useful in enriching your domain model. However, the use of any of these capabilities is optional - it is not necessary for a domain model project to have a reference to this assembly.

#### NakedObjects

* DateTimeExtensions. Useful methods that will automatically be added to instances of DateTime for comparing dates, deliberately ignoring the time element, for example: IsBeforeToday()
* ViewModel. A ready-made implementation of IViewModel, for view models based on a single Root type. See [View Model](#view_model).
* IHasGuid. This interface (along with several others below) is intended for use with 'Polymorphic Associations' (See [How to handle associations that are defined by an interface rather than a class](#interface_association)) and specifically to work in conjunction with NakedObjects.Services.ObjectFinder. If the class being associated implements IHasGuid, then the compound key will use this Guid (together with the fully qualified type name) to form the compound key. This has the advantage that the Guid may be set up when the object is created rather than waiting until the object is persisted (if the keys are database generated, that is). This is important when defining interface associations between transient objects that are all persisted in one transaction.
* IHasIntegerId. Merely defines that object has a single integer key called Id. Used by PolymorphicNavigator, for example.
* IPolymorphicLink. Designed to work with the PolymorphicNavigator service (below).
* PolymorphicLink. A ready-made implementation of IPolymorphicLink.
* ReasonBuilder. Helper object for constructing a string reason to be rendered to the user within, say, a Validate or Disable method. Separates multiple appended reasons with semi-colons.

#### NakedObjectsServices

* AbstractFactoryAndRepository. Convenience super class for factories and repositories that wish to interact with the container.
* IKeyCodeMapper. Defines a service that can convert between a key and a string code. Possible uses include: key encryption, custom key separators.
* IObjectFinder. Defines a mechanism for retrieving a domain object given a 'compound key' (a single string that defines both the type and the identity of that object). Also provides a method for determining the compound key for a given object. The intent of making both methods templated, is that it allows for the possibility of different types (or perhaps different namespaces of types) having different ways of putting together the compound key.
* ITypeCodeMapper. Defines a service that can convert between a Type and a string code where you don't wish to use the fully-qualified type name as the string representation. Possible uses include: to create compound keys for defining polymorphic associations;To create Oids for use in URLs.
* ObjectFinder. An implementation of IObjectFinder. Works with multiple keys, of type Integer, String, Short, or Char
* ObjectFinderWithTypeCodeMapper. An implementation of IObjectFinder that will delegate the string representation of a type to an injected ITypeCodeMapper service, if one exists. (Otherwise it will default to using the fully-qualified class name).
* PolymorphicNavigator. See [How to handle associations that are defined by an interface rather than a class](#interface_association).
* PolymorphicNavigatorWithOid. An alternative implementation of the PolymorphicNavigator designed to work specifically with implementations of IPolymorphicLinkWithOid.
* SimpleRepository. This is a simple, typed, implementation of a repository that can be useful when writing tests.

#### NakedObjects.Utils

* KeyUtils. Utility methods for obtaining and making use of domain object keys, whether explicitly defined, or inferred by convention.
* NameUtils. Utility methods for manipulating names of domain model elements - for use in presentation, for example.
* TypeUtils. Utility methods for safely obtaining and using types defined within a domain model.

# A how-to guide

## Using the Naked Objects IDE

The Naked Objects 'IDE' consists of a set of ItemTemplates and Code Snippets designed to boost productivity and coding standards when creating domain models for running with Naked Objects. They are installed via the NakedObjects.Ide NuGet package, which is automatically installed with the NakedObjects.ProgrammingModel package.

### Item Templates

Item templates are installed here: ..\Visual Studio [VS version]\Templates\ItemTemplates\C#\Naked Objects

If the templates are not appearing for you - check that your Visual Studio directory does already have the \Templates\ItemTemplates\ sub-directories. Sometimes problems arise when people have been using a Beta version of Visual Studio and then upgraded to the released version - with the result that they do not have the correct directory structure.

The templates will be offered, within a Naked Objects folder, whenever you select Add > New Item > Naked Objects. Here is a list of the current templates:

* **DomainObject** Creates a new domain model class with a ready-made Id property, and standardised code regions
* **DbContext** Creates a new class inheriting from DbContext
* **DbMapping**. Creates a new class for specifying database mappings in Code First.
* **DbInitialiser**. Creates a new class for initialising the database upon model changes in Code First.
* **Fixture** Creates an [object fixture](#_Using_object_fixtures) class for use within XATs.
* **Repository** Creates a new repository (and/or factory) class inheriting from AbstractFactoryAndRepository.
* **XAT** creates a new class to contain XATs.
* **CodeFirstReverseEngineerTemplate**. See [Using the Entity Framework Power Tools to reverse engineer a database into code](#_Using_the_Entity).

### Code Snippets

The Code Snippets will have been installed here: ..\Visual Studio [VS version]\Code Snippets\[Language]\My Code Snippets\NakedObjects

If the snippets are not appearing for you - check that the your Visual Studio directory does already have the \Templates\ItemTemplates\ sub-directories. Sometimes problems arise when people have been using a Beta version of Visual Studio and then upgraded to the released version - with the result that they do not have the correct directory structure.

#### Snippets for creating Actions

The following code snippets are relevant to defining an action, either within a Domain Object or a Service:

Table 1. Summary of code Snippets relevant to actions

| Name | Shortcut | Description |
| --- | --- | --- |
| Action | act | Adds a method intended to be a user action. |
| Action - default parameter | actdef | Adds a method to specify the default value for a parameter on a corresponding action method. |
| Action - choices | actcho | Adds a method to specify the choices (drop-down lists) for a parameter of a corresponding action method. |
| Disable | dis | Adds a method to disable a corresponding property or action dynamically. |
| Hide | hide | Adds a method to hide a corresponding property or action dynamically. |
| Validate | val | Adds a method to validate the value(s) being entered into a corresponding property or as parameters for a corresponding action. |

#### Snippets for creating Properties or Collections

The following code snippets are relevant to defining a property within a Domain Object:

Table 2. Summary of code Snippets relevant to properties

| Name | Shortcut | Description |
| --- | --- | --- |
| Property - Virtual | propv | Adds a virtual property (C#). |
| Property - choices | propcho | Adds a method to specify the choices (drop-down list) for a corresponding property. |
| Property - default | propdef | Adds a method to specify the default value for a corresponding property. |
| Disable | dis | Adds a method to disable a corresponding property or action dynamically. |
| Hide | hide | Adds a method to hide a corresponding property or action dynamically. |
| Modify | mod | Adds a method to intercept a user modification to a corresponding property. |
| Validate | val | Adds a method to validate the value(s) being entered into a corresponding property or as parameters for a corresponding action. |
| Collection | coll | Adds a collection, with actions for adding and removing objects. |
| Polymorphic Property | polyprop | Adds a property that is defined by an interface with many potential implementations (implementation delegates to an injected PolymorphicNavigator) |
| Polymorphic Collection | polycoll | Adds a collection that is defined by an interface with many potential implementations (implementation delegates to an injected PolymorphicNavigator) |

#### Snippets for creating or retrieving objects

The following code snippets for retrieving objects are intended for use either within a domain object or a service.

Table 3. Summary of code Snippets relevant to retrieving objects

| Name | Shortcut | Description |
| --- | --- | --- |
| Factory Method | fact | Adds a factory method that returns a new instance of a specified type. |
| New Transient Instance | newt | Code to create a new transient instance of a specified class. |
| Find Query | find | Adds a query method to find a single matching object. |
| List Query | list | Adds a query method to return a list of matching objects. |

#### Other Snippets

Table 4. Summary of other code Snippets

| Name | Shortcut | Description |
| --- | --- | --- |
| Injected Service | injs | Adds a property containing a reference to a Service, to be injected automatically when the object is instantiated. |
| Injected Container | injc | Adds a property containing a reference to an IDomainObjectContainer, to be injected automatically when the object is instantiated. |
| Title | title | Adds a Title() method to function as the domain object's title. |
| Icon Name | icon | Adds an IconName() method to specify the icon manually |

## The object life-cycle

Domain objects exist in one of two states: transient or persistent. A transient object exists only in memory: it is not known-to or managed-by the object Persistor. A persistent object is known-to and managed by the object store. A common scenario is that a new object is created in a transient state and returned to the user interface - where it appears in Edit mode, ready for the user to complete any mandatory fields before hitting the Save button, which changes the state of the object to persistent.

Important: Once an object has been made persistent, it remains in a persistent state - it never goes back to being transient. If the user edits an already persistent object then hitting the Save button will then cause the persistent object to be updated with the changes made. It is not correct to say that the object is being persisted again.

A second scenario is where an object is created in a transient state, has its mandatory fields set up and is then persisted within the application code before being made available to the user.

A third, less common, scenario is a domain object that only ever exists in a transient state and is never persisted.

### How to create an object

When you create any domain object within your application code, the Naked Objects framework must be made aware of the existence of this new object - in order that it may subsequently be persisted, and/or in order that any services that the new object needs are injected into it. Just specifying New Customer(), for example, will create a Customer object, but that object will not be known to the framework.

The correct way to create an object within your application code is to invoke the NewTransientInstance method on the Domain Object Container. For example:

Customer newCust = Container.NewTransientInstance<Customer>;

The new object will have been created in a transient state. It may be returned to the user to be completed and persisted, or persisted explicitly within your code.

Warning: It is possible to create a transient object within another transient object, but you need to be careful. When the framework persists any transient object, it will automatically persist any other transient object referenced by that object - so you will need to ensure that they all exist in a valid state before persisting any of them. Moreover, if any of these transient objects are to be exposed to the user (while in their transient state), then you need to anticipate the fact that the user could elect to save any of the transient objects at any point - which could cause the graph of related objects to be persisted in an invalid state.

The recommended approach is, if possible, not to permit the user to create a new transient object that is a child of an existing transient object, but, rather, to require the user to save the parent object first. This can be done by marking-up actions that create child objects with Disabled(WhenTo.UntilPersisted).

If wish to trigger some additional behaviour when the object is created by code outside of the object see [How to insert behaviour into the object life cycle](#_How_to_insert).

### How to persist an object

There is no need to write any special code for persisting an object. When a new transient is returned to the user then the user is provided automatically with a Save button which will change the object to a persistent state (assuming all mandatory fields have been completed).

If you wish to persist an object within your code invoke the NewTransientInstance method on the Domain Object Container. For example:

Customer newCust = Container.NewTransientInstance<Customer>;

newCust.Name = "Charlie";

Container.Persist<Customer>(ref newCust);

If you wish to trigger some additional behaviour when the object is persisted (by the user or by code outside of the object) see [How to insert behaviour into the object life cycle](#lifecycle_events).

### How to update an object

There is no need to write any special code for updating an object. If the user edits a persisted object then the changes will be made automatically when they save their changes. And if the programmer changes a property on an object then the changes will be notified to the object Persistor automatically.

If you wish to trigger some additional behaviour when the object is updated (by the user or by code outside of the object) see see [How to insert behaviour into the object life cycle](#lifecycle_events).

### How to delete an object

As with creating an object, if you wish to delete an object that is already persistent then you must notify the framework via the DisposeInstance method on the Domain Object Container:

Container.DisposeInstance(persistentObject)

Note: When working with the Entity Object Store, this method will delegate much of the work to the Entity Framework's delete functionality. As well as deleting the appropriate row(s) from the database table(s), Entity Framework will attempt to delete any persisted references to the object held in other objects - in other words will attempt to delete Foreign Keys to the deleted rows. If any of those Foreign Key columns are non-nullable in the database, then an error will be thrown and the whole transaction rolled back. In such circumstances, the programmer must take responsibility to delete the associated objects first - though this can be as part of the same transaction. For example, if you want to delete a persisted Customer object, but the Order object has a non-nullable reference to a Customer, then the method for deleting the Customer should first delete any Orders associated with that Customer.

### How to retrieve existing instances

If you need to retrieve instances from within a method on a domain object then you have two options:

* If a suitable method exists on a Repository then just inject that Repository into the domain object and call the method.
* Call the Instances method on an injected Domain Object Container. The method returns an IQueryable of the required object type, on which you may then invoke LINQ queries.

### How to insert behaviour into the object life cycle

See [LifeCycle methods](#_LifeCycle_methods).

The following are some examples of using these those life-cycle methods:

* Using the Created method to set the object into an initial state.
* Using the Loaded method to calculate the total, if you don't wish to have that total persisted explicitly.
* Using the Updating method to change a version property (e.g. LastUpdated) on an object just before it is persisted. Note, however, that you should update the private variable (e.g. myLastUpdated) not the public property - as the latter would initiate a new call to Updating and result eventually in a Stack Overflow Error!

### How to specify that an object should never be persisted

Use the standard Entity Framework Code First attribute: System.ComponentModel.DataAnnotations.Schema.NotMapped.

(If the object is intended for display to the user you should use the [View Model](#view_model) pattern).

### How to specify that an object should not be modified by the user

Use the Immutable attribute.

### How to specify that a class of objects has a limited number of instances

Use the Bounded attribute. A common way of describing this is that the whole (limited) set of instances may be rendered to the user as a drop down list - but the actual interpretation will depend upon the form of the user interface.

### How to implement concurrency checking

See [How to handle concurrency checking](#_How_to_handle).

## Object presentation

### How to specify a title for an object

A title is used to identify an object to the user in the user interface. For example, a Customer's title might be the organization's customer reference, or their name. The simplest way to specify a title for a domain object is to add a title attribute to one of the value properties (usually a string - but it may be any type of property). For example:

public class Employee {

[Title()]

public virtual string Name() { get; set;}

...

}

If you wish to construct the title from more than one property then you may provide a Title method, that returns a String.

A recommended practice is to use a NakedObjects.ITitleBuilder object to build the title. You may obtain this by calling Container.NewTitleBuilder(). ITitleBuilder works much like StringBuilder, but provides a number of useful methods to help in the construction of titles. The Append method will add a property to a title with a space in between; Concat adds the property without a space. There are overloaded versions of each method, that provide various formatting options, including the use of joiners (such as punctuation) and formatting strings for use when adding dates and other values to which standard formatting strings may be applied. If the property being added has a null value, then that Append or Concat statement will be ignored. For example, the following code would produce a title of the form:

Joe Bloggs, 07-Jan-63

public string Title()

{

var t = Container.NewTitleBuilder();

t.Append(FirstName).Append(LastName).Append("," DateOfBirth, "dd-MMM-yy");

return t.ToString();

}

Use the Title snippet (shortcut title) for creating this method.

If there is no title attribute on an object, and no title method, then the framework will use the object's ToString() method as the title. This means that in the above example, you could choose to rename the above method from Title to ToString (overriding the inheriting ToString()) and it would work the same way.

The recommended best practice is to construct titles from value fields within the objects, such as strings and dates. It is OK to include reference properties within a title, indeed the ITitleBuilder.Append method is designed to cope with this. However, you should be aware that if you include one or more reference properties within your title, then this will force those properties to be resolved when the object is loaded - instead of being resolved lazily as they are needed. This all happens transparently and is not usually a problem. However, it could slow down the performance of your application.

### How to control the layout of an object’s action menu

See [Object Menus](#_Object_Menus).

### How to specify a name and/or description for an object

By default, the name (or type) of an object, as displayed to the user will be the class name. However, if a DisplayName attribute is included, then this will override the default name. This might be used to include punctuation or other characters that may not be used within a class name.

By default the framework will create a plural version of the object name by adding an 's' to singular name. For irregular nouns or other special case, the Plural attribute may also be used to specify the plural form of the name explicitly.

(Note that there is an entirely separate mechanism for dealing with Internationalisation, which is described elsewhere).

The programmer may optionally also provide a Description attribute, containing a brief description of the object's purpose, from a user perspective. The framework will make this available to the user in a form appropriate to the user interface style - for example as 'balloon' help.

### How to specify that an object should be always hidden from the user

Use the Hidden attribute.

## Properties

The following conventions are concerned with specifying the properties of an object, and the way in which users can interact with those properties.

### How to add a property to a domain object

Properties can be 'auto properties' but they must be marked virtual. The simplest way to add a property is using the propv code snippet (standing for 'property - virtual').

### How to prevent the user from modifying a property

Preventing the user from modifying a property value is known as 'disabling' the property.

To disable a property always, use the Disabled attribute.

To disable a property under certain conditions, use a Disable method. The syntax is:

public string Disable<PropertyName>()

A non-null return value indicates the reason why the property cannot be modified. The framework is responsible for providing this feedback to the user. For example:

public class OrderLine {

public virtual int Quantity { get; set;}

public string DisableQuantity() {

if (HasBeenSubmitted()) {

return "Cannot alter any quantity after Order has been submitted";

}

return null;

}

}

The NakedObjects.ReasonBuilder class may be used to construct the message. The Reason property on ReasonBuilder will return the message as a string; if no message has been added, it will return a null value, as shown below:

public class OrderLine {

public virtual int Quantity { get; set;}

public string DisableQuantity() {

var rb = new ReasonBuilder();

rb.AppendOnCondition(HasBeenSubmitted(), "Cannot alter any quantity after Order has been submitted");

return rb.Reason;

}

}

Use the Disable snippet (shortcut dis) for creating this method.

To apply the same rules to all the properties on an object - for example to disable all the properties once the object has been persisted - create a method DisablePropertyDefault that returns a string, just as for an individual disable-property method. For example:

public string DisablePropertyDefault() {

if (Container.IsPersistent(this)) {

return "Cannot edit property once the object is saved";

}

return null;

}

If you wish to apply a rule to most, but not all of the properties then you may add the DisablePropertyDefault method, and then provide individual Disable methods for properties where you wish to override this default behaviour.

Having a DisablePropertyDefault method on a class and then using the Disable attribute on individual properties is not recommended - as the behaviour is not guaranteed.

### How to make a property optional (when saving an object)

Use the [[Optionally](#_Optionally)] attribute.

### How to specify the size of String properties

Use the StringLength, TypicalLength and MultiLine attributes.

### How to validate user input to a property

To validate that an input falls within a specific range, using the RangeAttribute.

To validate that an input value conforms to a particular format, use the Mask or RegularExpression attributes.

For more complex forms of validation, use a Validate method, for which the syntax is:

public string Validate<PropertyName>(object value)

If the proffered value is deemed to be invalid then the property will not be changed. A non-null return String indicates the reason why the member cannot be modified/action be invoked. The framework is responsible for providing this feedback to the user. For example:

public class Exam {

public virtual int Mark { get; set;}

public string ValidateMark(int mark) {

if (! (mark >= 0 & mark <= 30)) {

return "Mark must be in range 0 to 30";

}

return null;

}

}

This example is intended to illustrate the syntax of a Validate method. If your validation logic is actually as simple as defining a range for a numeric value, then you can just use the Range attribute instead of a Validate method.

The NakedObjects.ReasonBuilder class may be used to construct the message. The Reason property on ReasonBuilder will return the message as a string; if no message has been added, it will return a null value, as shown below:

public string ValidateMark(int mark){

ReasonBuilder rb;

rb.AppendOnCondition(! (mark >= 0 & mark <= 30), "Mark must be in range 0 to 30");

return rb.Reason;

}

Use the Validate snippet (shortcut val) for creating methods like this.

### How to validate user input to more than one property

Sometimes you need to be able to validate more than one property together. For this purpose you can use a Validate method that takes multiple parameters. This may be used in conjunction with individual validate methods on any or all of the properties. The classic example is having two date properties, where the FromDate cannot be after the ToDate, as shown in the following example:

public virtual DateTime FromDate { get; set; }

public string ValidateFromDate(DateTime d) {

if (!d.IsAfterToday()) {

return "Must be after Today";

}

return null;

}

public virtual DateTime ToDate { get; set; }

public string Validate(DateTime fromDate, DateTime toDate) {

if (fromDate.Date > toDate.Date) {

return "From Date cannot be after To Date";

}

return null;

}

In this example the FromDate property has a corresponding ValidateFromDate method to ensure that the date is after today. The ToDate property has no corresponding method - though it could have if, for example, you wanted to limit it to the next 12 months. The other Validate method is concerned with the relationship between the two properties. Note that this method is 'connected' to the two properties by dint of the specific names used for the parameters - writing the method as Validate(DateTime d1, DateTime d2) would result in the method being ignored as there are no Date properties called D1 or D2. You can thus have several of these multi-property validation methods on an object - each addressing a different set of properties. In theory a single property can participate in multiple such validation methods (as well as its own individual validation method); in practice such an approach would likely lead to much confusion and be difficult to debug.

A multi-property validation method will only be called once each of the properties is itself in a valid state. In the above example, both the FromDate and ToDate properties are mandatory (because they have not been marked up as Optionally). So the Validate method will only be called when both dates have been entered correctly. This means that, for example, there is no need to check for null values. The (string) message returned by the validate method (if validation has failed) will be displayed next to the property that has just been entered.

### How to specify a default value for a property

If your property is a value type (the user types in text) and the required default value may be statically defined, then you can just use the DefaultValue attribute on the property.

If your property is a reference type (another domain object) or you wish to specify a default value dynamically, the you can create a Default method, for which the syntax is:

public <Property type> Default<PropertyName>()

For example:

public class Order

{

public virtual Address ShippingAddress() Address { get; set;}

public Address DefaultShippingAddress()

{

return Customer().NormalAddress();

}

}

Use the Property - default snippet (shortcut propdef) for creating this method.

Value properties (such as dates and numbers) will, by default, show up on a transient object as blank fields. If you want a non blank field then you need to create a Default method as shown above. This is deliberate - so that the default behaviour is that the user is forced to enter a value.

If you create a transient object programmatically and set any value on that object within the same method then this will override any default value - as you would expect.

### How to specify a set of choices for a property

See also [How to handle enum properties](#enums) and[How to specify auto-complete for a property](#auto-complete_property) .

The simplest way to provide the user with a set of choices for a property (possibly rendered as a drop-down list, for example) is to ensure that the type used by the property is marked with the Bounded attribute - which will result in all instances of that type being offered to the user as a set of choices (typically as a drop-down list). If you wish to present the list with a sub-set of these, or with another customised set of choices - for example the set of all the Addresses known for a particular Customer - then you can write a Choices method:

And for specifying a list of choices is:

public <array or collection of property type> Choices<PropertyName>()

The full code for our example above is:

public class Order {

public virtual Address ShippingAddress() Address { get; set;}

public List<Address> ChoicesShippingAddress() {

return Customer().AllActiveAddresses();

}

}

Use the Property - default snippet (shortcut propdef) and the Property - choices snippet (shortcut propcho) for creating these methods.

#### Conditional Choices

It is possible to specify a set of choices for a property based on the selection(s) already made for another property or properties - for example to vary the available choices for a Province property based on the Country selected. We refer to this pattern as 'Conditional Choices'. An example is shown below:

public class Address {

public virtual Country CountryOfResidence {get; set;} //Where Country is a [Bounded] class

public virtual Province Province {get; set;}

public IList<Province> ChoicesProvince(Country countryOfResidence) {

if (country == null) { return new List<Province>; }

var q = from p in Container.Instances<Province>()

where p.Country.Id == countryOfResidence.Id

select p

return q.ToList();

}

}

In the above example code the selected value for the CountryOfResidence property is passed in as a parameter to the ChoicesProvice method. Note that the parameter name countryOfResidence must exactly match the property name CountryOfResidence, except for the character case, and the types (Country) must also match. Note also that the code guards against being called with a null value, returning an empty set of choices in this case (it could also return a default set of choices).

If used on a transient object, any values on which the set of choices depends must be passed in as parameters to the Choices method: you cannot refer to property values directly.

### How to specify auto-complete for a property

A common pattern in a web application is to allow the user to start typing a string and to provide the user with a dynamically-generated drop-down list of matches. This can be achieved using an AutoComplete[PropertyName]recognised method. This may be applied to a string property, but is typically most useful in the context of reference objects, as in the example below:

public virtual Customer ForCustomer { get; set; }

[PageSize(10)]

public IQueryable<Customer> AutoCompleteForCustomer( [MinLength(3)] string name) {

return CustomerRepository.FindCustomerByName(name);

}

When Naked Objects detects a matching AutoCompleteXxx method, as above, when the object is in Edit mode the user will be given a text field in which a string may be typed. The user will be presented with a dynamically-generated drop-down list of object titles, based on the IQueryable<T> returned by the method. Note that is is up to the implementation of the method how the match is performed - for example whether the match is a 'starts with' or 'contains' and whether or not it is case-sensitive.

The PageSize attribute specifies the number of matches to be presented to the user (there is no ability to page through more matches, however). If no PageSize attribute is added then the default page size for the system will be used - to avoid the risk of returning a very large number of matches.

The MinLength attribute is also optional - this specifies the minimum number of characters that the user must provide before an attempted match is made. If no MinLength attribute is specified, the search will be initiated on entering a single character.

Note: For a reference property, the return type of the AutoComplete method must be either IQueryable<T> where T is the type of the property, or just T (i.e. a single matching object). For a string property only, the return type of the AutoComplete method may be IEnumerable<string>.

### How to set up the initial value of a property programmatically

Initial values for properties may be set up programmatically within the created() method on the object. (See [The object life-cycle](#_The_object_life-cycle)).

### How to trigger other behaviour when a property is changed

If you want to invoke functionality whenever a property is changed by the user, then you should create a modify<propertyName> and include the functionality within that. For example:

public virtual int Amount { get; set;}

public void ModifyAmount(int newAmount)

{

Amount = newAmount;

AddToTotal(newAmount);

}

Use the Modify snippet (shortcut mod) to create this method.

The reason for the ModifyAmount method is that it would not be a good idea to include the AddToTotal call within the property's set method, because that method may be called by the persistence mechanism when an object is retrieved from storage.

You may optionally also specify a Clear<PropertyName> which works the same way as modify Modify <propertyName> but is called when the property is cleared by the user. Clear<PropertyName> does not take any parameters.

If the value of a property is changed by the user, from one non-null value to another non-null value, then the framework will first call the Clear<PropertyName> method (if it exists) and will then call the Modify<PropertyName> (if it exists) with the new value.

### How to control the order in which properties are displayed

Use the MemberOrder attribute.

### How to specify a name and/or description for a property

#### Specifying the name for a property

By default the framework will use the property name itself to label the property on the user interface. If you wish to over-ride this, use the DisplayName attribute on the property.

#### Specifying a description for a property

Use the Description attribute on the property itself.

The framework will take responsibility to make this description available to the user, for example in the form of a 'balloon help'.

### How to hide a property from the user

To hide a property always: use the Hidden attribute.

To hide a property under certain conditions use a Hide method. The syntax is:

public bool Hide<PropertyName>()

A true return value indicates that the property is hidden. For example:

public class Order {

public virtual string ShippingInstructions { get; set;}

public bool HideShippingInstructions()

{

return hasShipped();

}

...

}

Use the Hide snippet (shortcut hide) to create this method.

To apply the same rules to all the properties on an object - for example to hide all the properties once the object has been persisted. To do this, just create a method HidePropertyDefault that returns a boolean, just as for an individual hide-property method. For example:

public bool HidePropertyDefault() {

return Container.IsPersistent(this);

}

If you wish to apply a rule to most, but not all of the properties then you may add the HidePropertyDefault method, and then provide individual Hide methods for properties where you wish to override this default behaviour.

Having a HidePropertyDefault method on a class and then using the Hidden attribute on individual properties is not recommended - as the behaviour is not guaranteed.

### How to make a property non-persisted

If the property has a get but no set method then the field is not only unmodifiable but will also not be persisted. This may be used to derive a property from other information available to the object, for example:

public class Employee

{

public virtual Department Department { get; set;}

//this is the derived property

public Employee Manager

{

get

{

if (Department == null)

{

return null;

}

else

{

return Department.Manager();

}

}

}

...

}

If you need to have a get and set for the property but do not wish to have that property persisted, use the NotPersisted attribute.

### How to handle File Attachments

A common requirement is to be able to attach a file to a domain object.

Viewing a file attachment will typically result in a temporary file being written to your client machine in order to launch the viewing application. This is common practice but may pose security issues for certain types of business application.

The simplest way is to use a byte array, which can store any binary file. Entity Framework will recognise a byte array property and persist it as a single Binary column. At the user interface, Naked Objects will present the property as a link, and clicking on the link will attempt to launch a separate viewer to display the contents of the byte array.

The byte array property will be disabled automatically - the user will not be able to edit it directly. However, if an action method contains a byte array parameter, then this will be rendered on the user interface with a Browse button, allowing the user to browse for a file to upload. Thus, for an attached file that the use may modify the code will look something like this:

public virtual byte[] Attachment { get; set; }

public void AddOrChangeAttachment(byte[] newAttachment) {

Attachment = newAttachment;

}

The use of a simple byte array to handle a file attachment has two limitations though:

* There is no ability to associate a file name with the content, so on the user interface the link will always be rendered as a generic **Show File** link.
* There is no associated MIME type, to give a hint of which type of viewer to launch.

To overcome these limitations, you have the option to use the FileAttachment type. The NakedObjects.Value.FileAttachment type (in NakedObjects.Helpers.dll) has properties for the attachment name, the MIME type, and the contents (as a byte array). The name is used as the text for the link on the user interface, and the MIME allows the system to determine how the content should be viewed. A broad range of MIME types can be viewed directly inside the browser: other types will result in the launch of a separate viewer.

The FileAttachment type is used for display purposes only, and will typically be derived from three separate persisted properties, which will typically be hidden from the user. As with byte array, you will need a separate action to allow the file attachment to be uploaded, as shown in the following example code:

public virtual FileAttachment Attachment {

get {

if (AttContent == null ) return null;

return new FileAttachment(AttContent, AttName, AttMime){ DispositionType = "inline" };

}

}

[NakedObjectsIgnore]

public virtual byte[] AttContent { get; set; }

[NakedObjectsIgnore]

public virtual string AttName { get; set; }

[NakedObjectsIgnore]

public virtual string AttMime { get; set; }

public void AddOrChangeAttachment(FileAttachment newAttachment) {

AttContent = newAttachment.GetResourceAsByteArray();

AttName = newAttachment.Name;

AttMime = newAttachment.MimeType;

}

//Alternatively:

//public void AddOrChangeAttachment(FileAttachment newAttachment, string withNewName)

//{

// AttContent = newAttachment.GetResourceAsByteArray();

// AttName = withNewName;

// AttMime = newAttachment.MimeType;

//}

By setting the 'disposition type' to "inline" ( { DispositionType = "inline" } in the above code), the file attachment will be opened within the browser window (assuming that your browser has a viewer component for the mime type of the attachment. This is an optional behaviour.

#### How to associate multiple file attachments

To associate multiple file attachments, you will need to define a domain entity type (e.g. Attachment) that wraps the functionality shown above, and hold a collection of these new entity types, with suitable action methods for creating new ones (from an uploaded byte array or FileAttachment) or deleting existing ones.

### How to display an image

Any byte array or FileAttachment property may contain an image, but it will be displayed as a link. To display an image within an object view, you need to use the NakedObjects.Value.Image type, from the NakedObjects.Helpers.dll. Image works in a very similar manner to FileAttachment (in fact it is a sub-type) but is specifically recognised by Naked Objects as indicating that the contents should be rendered as an in-line image. The following shows a typical example of how to add an image property - to be displayed in-line - within an object.

public virtual Image Photo {

get {

return new Image(PhotoContent, PhotoName, PhotoMime);

}

}

[NakedObjectsIgnore]

public virtual byte[] PhotoContent {get; set;}

[NakedObjectsIgnore]

public virtual string PhotoName {get; set;}

[NakedObjectsIgnore]

public virtual string PhotoMime {get; set;}

public void AddOrChangePhoto(Image newImage) {

PhotoContent = newImage.GetResourceAsByteArray();

PhotoName = newImage.Name;

PhotoMime = newImage.MimeType;

}

//Alternatively:

//public void AddOrChangePhoto(Image newImage, string withNewName)

//{

// PhotoContent = newImage.GetResourceAsByteArray();

// PhotoName = withNewName;

// PhotoMime = newImage.MimeType;

//}

### How to handle enum properties

Naked Objects can use Enums - either for properties or for action parameters. There are two patterns for doing this (shown here for properties).

If your project is built to .NET 4.5 or above then the simplest and best option is to use the Enum as the property type, as shown below:

public Sexes Sex {get; set;}

...

public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}

At the user interface, the property will be displayed with the corresponding Name from and in Edit mode will be presented as a drop-down list. Note that the Names will be formatted using the same logic as class- and method-names in Named Objects, so that NotSpecified in the above example will be presented as Not Specified on screen. The options will be presented in alphabetical order: if you need to specify a different order, you may do this in a corresponding Choices method. (You may also specify a default value for the property).

The pattern above will not work with .NET 4.0, but you may use the alternate pattern shown below.

The second is to define an integer (or other 'integral type' such as a short, long, or byte) and then use the System.ComponentModel.DataAnnotations.EnumDataType to declare the Enum type that it corresponds to, as shown in this example:

[EnumDataType(typeof(Sexes))]

public int Sex {get; set;}

...

public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}

When using this pattern, a corresponding Choices or Default methods should return the same type as the property (an integer in the example above).

### How to work with date properties

(Note: NOF 8 handles date properties and parameters slightly differently to previous versions, as described below.)

A property of type DateTime in the domain code will be represented on the user interface according to the following rules.

1. By default, a DateTime property will be displayed to the user as a ‘date only’, with the time element ignored. The date will have the same value *irrespective of the timezone of the client (browser)*. The default format is d MMM yyyy (e.g. 3 Jul 2016). If you use a mask to forces the display of a time, then this will always be shown as zero, because the time component is not transmitted between client and server.
2. To force a DateTime property to behave as a ‘date + time’ then the property should be annotated with [DataType(DataType.DateTime)]. The default format is then d MMM yyyy hh:mm:ss (e.g. 3 Jul 2016 16:45:07). *The value of a ‘date + time’ property will be converted from UTC to the time-zone of the client (browser) before display*. Properties of this form are intended as read-only fields - for example to show when an object was created or last updated. There is no convenient mechanism for entering a date and time in a single field. If your domain model requires the capturing of a date and time (for example on an Appointment object) it is generally recommended that you manage the date and time elements in separate properties.
3. If a DateTime property has been specified as a ‘concurrency check’ field then, by default, it will be rendered as a ‘date + time’ - there is no need to add the [DataType(DataType.DateTime] attribute (though no harm if it is added). To override this behaviour and force display as date-only, annotate with [DataType(DataType.Date].

## Collection properties

This section defines patterns and practices that are specific to properties that represent collections of domain objects. For the definition of what Naked Objects recognises as a collection, see [Recognised Collection types](#_Recognised_Collection_types).

### How to add a collection property to a domain object

Collections should be defined by the generic System.Collections.Generic.ICollection interface but will need to be initialised with a concrete type such as List. (The initialisation should be done locally, as shown below, not in the object’s constructor). The property must be marked virtual. The simplest way to add a collection is using the coll code snippet.

The following example shows an Order object containing a collection of OrderLine objects:

public class Order {

...

private ICollection<OrderLine> myLines = new List<OrderLine>();

public virtual ICollection<OrderLine> Lines {

get {

return myLines;

}

set {

myLines = value;

}

}

}

Naked Objects does not support multiple associations of value types (such as strings, numbers). This is not considered to be a significant constraint as a collection of values is not a common modelling pattern, and considered bad practice by some modellers - who suggest that any such collection should be implemented as domain (entity) objects. If you need to use a collection of value types for programmatic purposes, it is recommended that you mark the collection as private or protected, to ensure that it is ignored by the framework. Note, however, that the framework can make use of lists of value types to provide a set of choices for a single value property.

### Adding-to or removing objects from a collection

In the standard NOF UI, the user may not directly add to or remove from a collection within an object in Edit mode: all collection properties are automatically disabled. If you want the user to be able to add to, or remove from a collection, then you should provide explicit actions to do this. The following shows an example:

public class Customer {

#region Vehicles (collection)

private ICollection<Vehicle> \_Vehicles = new List<Vehicle>();

public virtual ICollection<Vehicle> Vehicles {

get {

return \_Vehicles;

}

set {

\_Vehicles = value;

}

}

public virtual void AddToVehicles(Vehicle value) {

if (!(\_Vehicles.Contains(value))) {

\_Vehicles.Add(value);

}

}

public virtual void RemoveFromVehicles(Vehicle value) {

if (\_Vehicles.Contains(value)) {

\_Vehicles.Remove(value);

}

}

public IList<Vehicle> ChoicesRemoveFromVehicles(Vehicle value) {

return Vehicles.ToList();

}

#endregion

}

Note the following:

The AddToVehicles and RemoveFromVehicles methods will show up in the user interface as actions. If the user invokes the RemoveFromVehicles action, the ChoicesRemoveFromVehicles method will provide the user with a drop-down list of all the existing vehicles in the collection.

If you use the coll snippet to add a collection into your object, then Add and Remove methods are automatically generated for you.

### How to create a derived collection

Collections can be derived, in the same way as properties. These are not persisted, but are represented as ReadOnly collections. For example:

public class Department

{

// Derived collection  
 [NotPersisted, NotMapped]

public ICollection<Employee> TerminatedEmployees

{

get

{

List<Employee> results = new List<Employee>();

foreach (Employee e in Employees)

{

if (e.IsTerminated())

{

results.Add(e);

}

}

return results;

}

}

...

}

If you are working Code First, you should mark up the derived collection with a NotMapped attribute. Otherwise, Entity Framework will attempt to map the collection to database relationship. Adding the NakedObjects.NotPersisted attribute is optional, but is marginally more efficient when executing.

### How to control the order in which table rows are displayed

To control, programmatically, the order in which rows are displayed, you can just incorporate the Linq OrderBy method into the get method for the collection. In the following example, the collection of SalesOrderDetails will be displayed (by default) ordered by the UnitPrice:

private ICollection<SalesOrderDetail> \_details = new List<SalesOrderDetail>();

public virtual ICollection<SalesOrderDetail> Details {

get {

return \_details.OrderBy((x) => x.UnitPrice).ToList();

}

set {

\_details = value;

}

}

If you use this pattern, however, be aware that the collection returned by the property's get is not the same as the underlying private collection. Therefore, when you are adding to or removing from the collection, it is important that you work with the underlying collection. The best way to do this is to ensure that you always work through the AddTo / RemoveFrom associated methods - which are automatically added when you create a collection using the coll code snippet.

### How to specify which columns are displayed in a table view

Use the TableView attribute.

## Actions

An 'action' is a method that we expect the user to be able to invoke via the user interface, though it may also be invoked programmatically within the object model. The following conventions are used to determine when and how methods are made available to the user as actions.

### How to add an action to an object

See [Action](#_Action).

### How to specify the layout of the menu of actions on an object

See [Object Menus](#_Object_Menus).

### How to define a contributed action

See [Contributed action](#contributed_action).

### How to prevent a service action from being a contributed to objects

If you want an action on a service to be available to the user on the service menu, but not contributed to object menus, use the [NotContributedAction](#notcontributed_attribute) attribute.

### Hor to specify parameter names and/or descriptions

As with properties, the framework will pick up parameter names reflectively and reformat these for presentation to the user. If you wish to override this mechanism and specify a different name then use the DisplayName attribute. This is especially useful if you wish to include punctuation or other characters that would not be permissible in a parameter name.

Similarly, any parameter may be given a short user-description using the Description attribute. The framework takes responsibility to make this available to the user.

### How to make a parameter optional

Use the [Optionally](#_Optionally) attribute.

### How to specify a default value for a parameter

When an action is about to be invoked, then default values may be specified for any or all of its parameters.

If the parameter is a value type (the user types in text) and the required default value may be statically defined, then you can just use the DefaultValue attribute on the parameter.

If your parameter is a reference type (another domain object) or you wish to specify a default value dynamically, them you can use a Default method.

The default value for each parameter is specified via a separate method, with parameters numbered from zero. You need only write such methods for those parameters where you require a default value. There are two alternative versions of the syntax is as follows:

public <parameter type> Default<ActionName>([<parameter type> parameterNameSameAsOnAction])

//OR

public <parameter type> Default<parameter number><ActionName>()

The second syntax may be used where the action has more than one parameter of the same type. Note that parameters are numbered from zero.

Each method returns a single value of the appropriate type for its corresponding parameter . The following code shows both forms of the syntax:

public class Customer

{

public Order PlaceOrder(Product product, int quantity, int promotionCode) {...}

public Product DefaultPlaceOrder()

{

return ProductMostRecentlyOrderedBy(this.getCustomer());

}

public int Default1PlaceOrder()

{

return GetQuantityFromPreviousOrder();

}

...

}

Use the Action - default snippet (shortcut actdef) to create this method.

### How to specify a set of choices for a parameter

See also [Enums](#enums).

See also [How to specify auto-complete for a parameter](#auto-complete_param).

Where the type of a parameter is annotated with Bounded, then the user will automatically be provided with each of the instances of that type in the form of a drop-down list or equivalent selection device. Sometimes, however, it is desirable to specify a set of choices that does not constitute a bounded set. This is a achieved by adding one or more Choices methods. You need only write such methods for those parameters where you wish to specify a set of choices other than by using a bounded set. The recommended is:

public List<parameter type> Choices<parameter number><ActionName>()

Note that parameters are numbered from zero.

Each such method returns a collection (or an array) of the same type as the corresponding parameter. (Note that parameters are numbered from zero). For example:

public class Customer {

public Order PlaceOrder(Product product, int quantity) {...}

public List<Product> Choices0PlaceOrder()

{

return LastFiveProductsOrderedBy(this.Customer());

}

public Product Default0PlaceOrder() {...}

...

}

As shown in the examples, above, you may specify Choices and a Default value for the same parameter.

Use the Action - choices snippet (shortcut actcho) to create this method.

It is possible to specify a set of choices for a parameter based on the selection(s) already made for another parameter or parameters - for example to vary the available choices for a Province property based on the Country selected, as shown below:

public Address CreateAddress(string line1 As String, Country country, Province province)

public IList<Province> Choices2CreateAddress(Country country) {

if (country == null) { return new List<Province>; }

var q = from p in Container.Instances<Province>()

where p.Country.Id == countryOfResidence.Id

select p

return q.ToList();

}

In the above example code the numeral 2 in Choices2CreateAddress indicates that this method provides the choices for parameter 2 in the CreateAddress method - which is the province parameter (parameters are numbered from zero). However, the selected value for the country parameter (in this example the Country class is assumed to be a Bounded set) is passed in to this Choices method. Note also that the code guards against being called with a null value, returning an empty set of choices in this case (it could also return a default set of choices).

### How to allow selection of multiple choices

If you want the user to be able to select multiple options from a list, then there are the following two options:

1. The parameter should be an IEnumerable of a domain object type and either:
   * that domain type is a bounded set
   * or a set of choices is provided via an action Choices method.
2. The parameter type is an IEnumerable of string or integer and a set of choices is provided via an action Choices method.
3. The parameter type is an IEnumerable of an enum

The first case is illustrated in the following example:

public void AddStandardComments(IEnumerable<string> comments) {

foreach (string comment in comments) {

Comment += comment + "\n";

}

}

public string[] Choices0AddStandardComments() {

return new[] {

"Payment on delivery",

"Leave parcel with neighbour",

"Send SMS on delivery"

};

}

### How to specify auto-complete for a parameter

A common pattern in a web application is to allow the user to start typing a string and to provide the user with a dynamically-generated drop-down list of matches. This can be achieved using an AutoComplete[parameterNumber][ActionName] recognised method. This may be applied to a string parameter, but is typically most useful in the context of reference parameters, as in the example below:

public virtual Order CreateNewOrder(Customer forCustomer) {...}

[PageSize(10)]

public IQueryable<Customer> AutoComplete0CreateNewOrder( [MinLength(3)] string name) {

return CustomerRepository.FindCustomerByName(name);

}

When Naked Objects detects a matching AutoCompleteXxx method, as above, the user will be presented with a dynamically-generated drop-down list of object titles, based on the IQueryable<T> returned by the method. Note that is is up to the implementation of the method how the match is performed - for example whether the match is a 'starts with' or 'contains' and whether or not it is case-sensitive.

The PageSize attribute specifies the number of matches to be presented to the user (there is no ability to page through more matches, however). If no PageSize attribute is added then the default page size for the system will be used - to avoid the risk of returning a very large number of matches.

The MinLength attribute is also optional - this specifies the minimum number of characters that the user must provide before an attempted match is made. If no MinLength attribute is specified, the search will be initiated on entering a single character.

Note: For a reference parameter, the return type of the AutoComplete method must be either IQueryable<T> where T is the type of the parameter, or just T (i.e. a single matching object). For a string parameter only, the return type of the AutoComplete method may be IEnumerable<string>.

### How to specify the length or format for text-input parameters

Use the StringLength, TypicalLength, MultiLine, Mask or RegularExpression attributes.

### How to obscure input text (e.g. for a Password)

Use the DataType attribute: [DataType(DataType.Password)].

### How to validate parameter values

To validate that an input falls within a specific range, using the RangeAttribute.

To validate that an input value conforms to a particular format, use the Mask or RegularExpression attributes.

For more complex forms of validation, use a Validate method, which may be applied to any of the parameters individually, or to the set of parameters as a whole.

To validate a single parameter, there are two alternative forms of syntax:

public string Validate<ActionName>(<parameter type> parameterNameSameAsOnAction)

//OR

public string Validate<ParameterNumber><ActionName>(<parameter type> anyOldName)

The first syntax is easier to read. The second syntax may be used where the action has more than one parameter of the same type (to avoid the possibility of ending up with two validate methods that have identical signatures). Note that parameters are numbered from zero.

In both cases, a non-null return String indicates the reason why the member cannot be modified/action be invoked, and the viewing mechanism will display this feedback to the user. The following example code shows both forms of the syntax in use:

public class Customer {

public Order PlaceOrder(Product p, int quantity

string purchaseOrderNumber, string comment)

{...}

public string ValidatePlaceOrder(Product p) {

if (p.IsOutOfStock()) {

return "Product is out of stock";

}

return null;

}

//Parameter number 2 is used to match up to 'purchaseOrderNumber'

public string Validate2PlaceOrder(string pon)

{

if (! pon.StartsWith("PO"))

{

return "Purchase Order Number must start with 'PO'";

}

return null;

}

}

You may also validate multiple parameters together in a single validate method. The parameter names should match those used in the action method.

public class Price {

public void SpecifyApplicability(DateTime fromDate, DateTime toDate) {...}

public string ValidateSpecifyApplicability(DateTime fromDate, DateTime toDate) {

if (toDate.Date < fromDate.Date) {

return "From date cannot be before To date";

}

return null;

}

}

Use the Validate snippet (shortcut val) to create this method.

### How to work with date parameters

(See also [How to work with date properties](#_How_to_work_1)). By default action parameters of type DateTime will be treated as having a date only, and the value entered will not be associated with any time zone. If you wish the user to specify a date and a time (for example in creating an Appointment) it is recommended that these elements are handled as separate parameters.

### How to specify conditions for invoking an action

#### Disabling an action based on the state of the object

There may be circumstances in which we do not want the user to be able to initiate the action at all - for example because that action is not appropriate to the current state of the object on which the action resides. Such rules are enforced by means of a Disable method.

The syntax is:

public string Disable<ActionName>(<parameter type> param)

A non-null return String indicates the reason why the action may not be invoked. The framework takes responsibility to provide this feedback to the user. For example:

public class Customer

{

public Order PlaceOrder(Product p, int quantity) {...}

public string DisablePlaceOrder(Product p, int quantity)

{

if (isBlackListed())

{

return "Blacklisted customers cannot place orders";

}

return null;

}

}

It is also possible to permanently disable an action using the Disabled attribute. One possible reason for doing this might be during prototyping - to indicate to the user that an action is planned, but has not yet been implemented.

Use the Disable snippet (shortcut dis) to create this method.

#### Disabling multiple actions on an object

You may wish to apply the same rules to all the actions on an object - for example to disable all the actions until the object has been persisted. To do this, just create a method DisableActionDefault that returns a string, just as for an individual disable-action method. For example:

public string DisableActionDefault() {

if (!Container.IsPersistent(this)) {

return "Cannot invoke this action until the object has been saved";

}

return null;

}

If you wish to apply a rule to most, but not all of the actions then you may add the DisableActionDefault method, and then provide individual Disable methods for actions where you wish to override this default behaviour.

Having a DisableActionDefault method on a class and then using the Disable attribute on individual actions is not recommended - as the behaviour is not guaranteed.

### How to control the order in which actions appear on the menu

Use the MemberOrder attribute.

### How to hide actions

To hide an action always use the Hidden attribute. (This is generally used where a public method on an object is not intended to be a user action).

To hide an action under certain conditions use a Hide method. The syntax is:

public bool Hide<ActionName>(<parameter type> param)

A true return value indicates that the action should not be shown. For example:

public class Order

{

public void ApplyDiscount(int percentage) {...}

public bool HideApplyDiscount()

{

return isWholesaleOrder();

}

...

}

Use the Hide snippet (shortcut hide) to create this method.

To apply the same rules to all the actions on an object - for example to hide all the actions until an object has been persisted. To do this, just create a method HideActionDefault that returns a boolean, just as for an individual hide-action method. For example:

public bool HideActionDefault() {

return !Container.IsPersistent(this);

}

If you wish to apply a rule to most, but not all of the properties then you may add the HideActionDefault method, and then provide individual Hide methods for actions where you wish to override this default behaviour.

Having a HideActionDefault method on a class and then using the Hidden attribute on individual actions is not recommended - as the behaviour is not guaranteed.

To hide an action for certain users or roles, see 'Properties: Hiding a property for specific users or roles'. The same technique can be applied to actions. However, the caveats apply.

### How to pass a message back to the user

Sometimes, within an action it is necessary or desirable to pass a message to the user, for example to inform them of the results of their action ('5 payments have been issued') or that the action was not successful ('No Customer found with name John Smith'). DomainObjectContainer defines two methods for this purpose:

public void InformUser(string message)

public void WarnUser(message As string)

### How to work with transactions

All action methods are automatically wrapped in a transaction by the Naked Objects framework - so there is usually no need to write any specific code to start or end a transaction. If an exception is thrown within an action method, and not caught within application code, this will automatically cause the transaction to be aborted.

If you wish to abort a transaction, without throwing an exception, then you can call the AbortCurrentTransaction method on the Container.

## Working with Entity Framework

Naked Objects uses Microsoft’s Entity Framework to persist domain objects in a database. Most developers who work with Entity Framework now use it in ‘Code First’ mode - and this is what we now use throughout this manual. The name is slightly misleading: you can use ‘Code First’ mode even when creating an application to work against an existing database – ‘Code First’ just means that your persistence is entirely defined in program code (typically C#). Naked Objects can, however, work equally well with Entity Franework in the older mode, where the entity model is defined in XML with a .edmx file.

This manual does not attempt to provide an introduction to Entity Framework Code First development - rather it just emphasises what you need to do to make your project work with Naked Objects. We therefore recommend that you gain some general familiarity with Entity Framework Code First development: there are numerous on-line tutorials, and we also strongly recommend the book Programming Entity Framework - Code First by Julia Lerman and Rowan Millar.

When copying any domain code examples from the book or on-line Code First tutorials mentioned, please remember the following basic rules:

* Naked Objects requires that all properties are virtual.
* Naked Objects requires that all collections are virtual, and are initialised (but not in a constructor). See [Collection properties](#_Collection_Property).
* Entity Framework Code First makes all properties optional in the database, unless specified as mandatory (using the Required attribute, or via the Code First Fluent API). However, at the user interface NOF treats all properties as *mandatory*, unless marked up with the Optionally attribute - as we believe that this is the safer default behaviour.

### Overriding the default database schema generation

By default, Entity Framework Code First creates the database schema by following a set of conventions, based on the class and property names. These convention-based schema may be over-ridden or enhanced, either by using Code First Data Annotations in the domain classes, or by means of the Code First Fluent API. The latter is invoked by creating one or more configuration classes (inheriting from EntityTypeConfiguration<T>) and referencing them from within the DbContext, as in the following example (quoted from Programming Entity Framework - Code First by Julia Lerman and Rowan Millar):

namespace DataAccessFluent {

public class DestinationConfiguration : EntityTypeConfiguration<Destination>

{

public DestinationConfiguration() {

Property(d => d.Name).IsRequired();

Property(d => d.Description).HasMaxLength(500);

Property(d => d.Photo).HasColumnType("image");

}

}

public class LodgingConfiguration : EntityTypeConfiguration<Lodging> {

public LodgingConfiguration() {

Property(l => l.Name).IsRequired().HasMaxLength(200);

Property(l => l.Owner).IsUnicode(false);

Property(l => l.MilesFromNearestAirport).HasPrecision(8, 1);

}

}

//...

public class BreakAwayContextFluent : DbContext {

public BreakAwayContextFluent(string name) : base(name) { }

public BreakAwayContextFluent() { }

public DbSet<Destination> Destinations { get; set; }

public DbSet<Lodging> Lodgings { get; set; }

//...

protected override void OnModelCreating(DbModelBuilder modelBuilder) {

modelBuilder.Configurations.Add(new DestinationConfiguration());

modelBuilder.Configurations.Add(new LodgingConfiguration());

// ...

}

}

}

### Using data fixtures with Code First

When working Code First, you can create data fixtures (to pre-populate the database with) via the Seed method on the Database Initializer. The following example is quoted from Programming Entity Framework - Code First by Julia Lerman and Rowan Millar):

namespace DataAccess {

public class DropCreateBreakAwayWithSeedData :   
 DropCreateDatabaseAlways<BreakAwayContext> {

protected override void Seed(BreakAwayContext context) {

context.Destinations.Add(new Model.Destination {   
 Name = "Great Barrier Reef" });

context.Destinations.Add(new Model.Destination { Name = "Grand Canyon" });

//...

}

}

}

1. This custom database initializer may be set within the OnModelCreating method on your DbContext.

### How to handle concurrency checking

Naked Objects provides full support for concurrency checking - such that before a user saves any edits, or invokes any action upon an object, the framework will check to see that no other user has changed the state of that object.

Any domain object that needs to make use of this capability, must have a 'version' property that is guaranteed to change each time the persisted state of that object is changed. This property may be of type DateTime (acting as a time stamp), a string, a numeric value or a byte array. It may be visible to, or hidden from, the user. The responsibility for updating the property when changes are persisted may be performed by the domain code, but will more commonly be performed by the database, by means of a trigger or a calculated column.

The property must be marked up with the ConcurrencyCheck attribute, as shown in the example below:

public class Employee {

...

[ConcurrencyCheck]

public virtual DateTime LastUpdated { get; set; }

}

If you have any inheritance within your domain model, then the ConcurrencyCheck attribute should be applied to a property on the top-most class of each hierarchy, and should not be duplicated within any sub-classes. (Sub-classes may have their own LastUpdated or similar properties for other purposes, but these do not play a role in concurrency checking.)

Note that there is no automatic concurrency checking on objects that are used as parameters for actions, nor on objects selected to be invoked by a [collection-contributed action](#collection_contributed_actions).

*Note: As of NOF 8, concurrency checking is not undertaken on the objects selected to be invoked with a ‘collection-contributed action’. This change was to make the behaviour consistent with other aspects of the system. For example, if an object is used as a parameter within an action there is no automatic concurrency checking on that object.*

### How to specify 'eager loading' of an object's reference properties

By default, Entity Framework uses lazy loading - if an object holds a reference to another object then that second object is retrieved from the database only when the property is accessed. However, if a user retrieves and displays an object then many of the object's reference properties will be accessed, and this will result in multiple round-trips to the database. If this results in unacceptable performance (often, it won't) then the answer is to force the object to load some or all of its associated objects in one trip - this is known as 'eager loading'.

This is done using the .Include method in your LINQ queries. With the Entity Framework CTP Microsoft has provided an extension method that makes the the .Include method much easier to work with. To use this, ensure that you have a reference to the CTP within your project. This may be found at:

C:\Program Files\Microsoft ADO.NET Entity Framework Feature CTP5\Binaries\Microsoft.Data.Entity.CTP.dll

(on a 64-bit machine that's C:\Program Files (x86)... )

The following code shows an example of eager loading within a query method - ensuring that the object referenced within the Customer's Pet property is loaded at the same time as the Customer:

using system.data.entity;

var petOwners = Container.Instances<Person>().Where(x => x.HasPet).Include(x => x.Pet);

### How to implement complex types

TODO: review in the context of NOF 8

The core Naked Objects framework works well with the full capabilities of Complex Types, as defined by Entity Framework. However, the ability to use Complex Types within your application does depend upon the 'viewer' that you are using with the core framework:

* The Naked Objects MVC user interface supports ordinary Complex Types, but does not support nested Complex Types (where a Complex Type has a property that is itself a Complex Type).
* The Restful Objects API does not currently support Complex Types at all (though this is might be added in a future release).

(As of NOF 7.0) When working Code First it is necessary to annotate the complex type class with the ComplexType attribute, *even if you have specified* *a complex type by means of the Code First Fluent API*. This is because Naked Objects needs to be able to detect that it is a complex type independently of Entity Framework.

### How to work with multiple databases

Naked Objects can cope with multiple databases. You will need to add the DbContext for each database in the EntityObjectStoreConfiguration. See [Creating a Model project from scratch](#_Creating_a_Model).

### How to work with multiple database contexts

Naked Objects provides very good support for working with multiple contexts as shown below:

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {

var config = new EntityObjectStoreConfiguration();

config.UsingCodeFirstContext(() => new ModelCContext());

config.UsingCodeFirstContext(() => new ModelDContext());

return config;

}

These various contexts may correspond to separate databases, or they may point to the same shared database(s).

At run-time, when an object is retrieved (whether explicitly via Container.Instances<T>, or just by navigation) Naked Objects transparently identifies which of the contexts the type resides in and instructs Entity Framework to retrieve the object from that context. This is a very powerful capability, and enables things like polymorphic association.

However, this pattern does carry an overhead, that grows with the number of contexts you have. This is because - due to a severe limitation in the design of Entity Framework - the only way to find out if a context knows about a given domain type is to ask for that type, and catch the exception thrown if that type is not known. Exceptions are relatively expensive in processing terms, so raising lots of them - from polling multiple contexts - can be slow. Naked Objects does cache the mapping of types to contexts, but this cache is generated progressively, as each type is encountered.

This overhead can be eliminated by explicitly associating types with their context, using the syntax below:

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {

var config = new EntityObjectStoreConfiguration();

config.UsingCodeFirstContext(() => new

ModelAContext()).AssociateTypes(ModelATypes);

config.UsingCodeFirstContext(() => new

ModelBContext()).AssociateTypes(ModelCContext.ModelBTypes);

…

return config;

}

private static Type[] ModelATypes() {

return new Type[] {typeof(Payment), typeof(Invoice), ... };

}  
…

The AssociateTypes method (on ContextInstaller, which is returned by Using...Context) takes as a parameter any method that returns an array of types. Each of these specified associations will be cached on the session. The methods defining the array of types could be local - as shown for ModelATypes() above - or as an external static method, for example, on the appropriate code first DbContext - as shown for ModelCTypes above.

Although it is sensible to prime the cache with all the types that you will be using, there is no requirement to do so. If the framework comes across a type for which it does not know the context, it will poll the contexts to find it, and then cache the association - albeit with the overhead mentioned previously.

As a further refinement to this performance optimisation, there is a method SpecifyTypesNotAssociatedWithAnyContext on the EntityObjectStoreConfiguration. This method only adds value where you have a domain class (typically an abstract class, though not necessarily) that is not itself an entity - and therefore unknown to Entity Framework - but from which one or more entity types inherit.

A simple example of this exists within our AdventureWorks sample project, where many domain classes inherit from AWDomainObject - which provides a small amount of functionality common to many classes. In ordinary operation, Naked Objects will work up the inheritance hierarchy, until it finds a class that is unknown to any DbContext, and then work with the next level down. This incurs the overhead of a thrown and caught exception mentioned earlier. To improve efficiency, the AdventureWorks sample project now contains this code:

public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {

var config = new EntityObjectStoreConfiguration();

config.UsingCodeFirstContext(() => new MyContext())  
 .AssociateTypes(AllPersistedTypesInMainModel)

.SpecifyTypesNotAssociatedWithAnyContext(() => new[] {typeof (AWDomainObject)});

return config;

}

If there was more than just the class AWDomainObject to which this applied, then the method may be called with a delegate, in a manner similar to AssociateTypes e.g.:

installer.SpecifyTypesNotAssociatedWithAnyContext(TypesToIgnore);

The EntityObjectStoreConfiguration has a RequireExplicitAssociationOfTypes, which is set to false by default. If set to true when initialised, then the framework will throw an uncaught exception if the persistor is asked for any domain type that has not either been associated with a specific DbContext (using AssociateTypes) or with no context (using SpecifyTypesNotAssociatedWithAnyContext).

### How to write safe LINQ queries

#### Don't use the equality operator on objects; test for equality on the value properties

Don't write:

public IQueryable<Product> FindProducts(ProductCategory category) {

return Container.Instances<Product>().Where(x => x.Category == category);

Write:

public IQueryable<Product> FindProducts(ProductCategory category) {

return Container.Instances<Product>().Where(x => x.Category.Id == category.Id);

#### Don't call any method on a domain object within a query; refer only to properties

Don't write:

public IQueryable<Product> ListDiscontinuedProducts() {

return Container.Instances<Product>().Where(x => x.IsDiscontinued());

Write:

public IQueryable<Product> ListDiscontinuedProducts() {

return Container.Instances<Product>().Where(x => x.Status == "Discontinued");

Note, though that you can call methods on System classes e.g. Trim().ToUpper() on string.

#### Don't navigate references on any objects passed into the query; pass in any such required indirect references as variables in their own right

Don't write:

public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {

return Instances<Order>().Where(x => x.SentTo.Id != cust.BillingAddress.Id);

Write:

public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {

Address billingId = cust.BillingAddress.Id;

return Instances<Order>().Where(x => x.SentTo.Id != billingId);

#### When doing a join, don't try to use Container.Instances<T>() more than once inside a query; define a separate IQueryable<T> outside the query

Don't write:

var q = from p in Container.Instances<Product>()

from c in Container.Instances<Customer>()

where ...

Write:

var customers = Container.Instances<Customer>();

var q = from p in Container.Instances<Product>()

from c in customers

where …

Or, for greater clarity:

var customers = Container.Instances<Customer>();

var products = Container.Instances<Product>();

var q = from p in products

from c in customers

where …

### How to handle associations that are defined by an interface rather than a class

Entity Framework requires that any association is defined by a type that a Class (whether concrete or abstract). It does not natively support the concept of an association that is defined by an Interface.

To overcome this limitation of Entity Framework, Naked Objects provides two patterns for achieving the goal: the 'result interface association' and the 'polymorphic association'.

#### Result interface association

This is the simpler case where a property is defined by an interface but there is expected to be just a single implementation of that interface. In such cases the Interface is not defining a role for multiple objects to play - it is simply defining a reduced view of an object that is returned as the result of a query, say, perhaps in order to hide other aspects of that object's implementation. The 'role interface association' pattern as described above would still work here, but it is overkill. The specific 'result interface association' pattern is simpler to code, and potentially faster in execution. (It also means that the relationship may be implemented at database level as a foreign key, which the role interface association does not permit.)

The example below shows how to code the pattern.

Use the Result Interface Association snippet (shortcut resultia) to help create this code.

public class Order {

[Hidden()]

public virtual int CustomerId {get; set;}

private ICustomer \_Customer;

[NotPersisted()]

public ICustomer Customer {

get {

if (\_Customer == null && CustomerId > 0) {

\_Customer = CustomerRepository.FindById(CustomerId);

}

return \_Customer;

}

set {

\_Customer = value;

if (value == null) {

CustomerId = 0;

}

else {

CustomerId = value.Id;

}

}

}

#### Polymorphic Association

Polymorphic Associations (PAs) - associations between objects where the type is defined by an ‘role’ interface rather than by a concrete or abstract class.

The following example is based on a Payment object that has two PAs. The first is a single association to a Payee, defined by the role interface IPayee. In the example there are two separate implementations of IPayee: Customer and Employer. In practice there may be many more. The second PA is a multiple association, to a collection of type IPayableItem. In this example there are two implementations of the role IPayableItem: ExpenseClaim and Invoice. The intent of the domain model is represented in this diagram:



Next we turn to the implementation of the PA - initially from a domain modelling perspective, as shown below:



Note the addition of two new domain classes: PaymentPayeeLink and PaymentPayableItemLink. Both of these inherit from a generic helper class - PolymorphicLink<TRole, TOwner> - which is defined in the NakedObjects.Helpers assembly, installed by the NakedObjects.ProgrammingModel package.

For this particular example, the two new classes do not add any properties or methods to the generic helper class, they are simply defined thus:

public class PaymentPayeeLink : PolymorphicLink<IPayee, Payment> { }

public class PaymentPayableItemLink : PolymorphicLink<IPayableItem, Payment>

However, these Link classes could, in principle, add custom properties and/or methods. A particular use of this might be to de-normalise some properties (such as a Title property or other summary information) from the associated object.

Working CodeFirst, each of these Link domain objects will result in an equivalent table in the database. This is shown in the database diagram below (table names have been shortened for clarity):



Note that each of the Link tables has a foreign key reference back to the ‘owner’ (in this case a Payment). In the case of PayeeLinks, the foreign key is also the Id - because it is a single association. Each Link table has two columns that, in combination, specify the associated object: AssociatedRoleObjectType and AssociatedRoleObjectId. The latter is always an integer (one constraint of this pattern is that all objects that participate in the PA must have a single integer Id property called ‘Id’ in the code (the naming convention in the database is not mandated). The AssociatedRoleObjectType is a string, and may either take the form of the fully-qualified type name (e.g. ‘MyApp.Customer’) or a standard abbreviation (e.g. ‘CUS’) - this is configurable and may even be mixed. These two items permit Naked Objects to retrieve the - corresponding object - using methods provided by the generic PolymorphicLink class (which, in turn, delegates to an injected service called PolymorphicNavigator, which is also included in the NakedObjectsProgramming model).

##### Using the code snippets

Almost all the domain code needed to implement this pattern is generated by the use of two code snippets that are installed with the NakedObjects.Ide package (which you should install at solution level):

* polyprop for a polymorphic property such as the payee in the example above.
* polycoll for a polymorphic collection such as the payable items in the example above.

You will need to [register](#_Registering_domain_services) the NakedObjects.PolymorphicNavigator as a service.

The code below shows code from the example that was generated entirely by the polyprop snippet:

#region Payee Property of type IPayee ('role' interface)

[Disabled]

public virtual Payment\_Payee\_Link PayeeLink { get; set; }

private IPayee \_Payee;

[NotPersisted]

public IPayee Payee {

get {

return PolymorphicNavigator.RoleObjectFromLink(ref \_Payee, PayeeLink, this);

}

Set {

\_Payee = value;

PayeeLink = PolymorphicNavigator.UpdateAddOrDeleteLink(\_Payee, PayeeLink, this);

}

}

public void Persisting() {

PayeeLink = PolymorphicNavigator.NewTransientLink  
 <Payment\_Payee\_Link, IPayee, PolymorphicPayment>(\_Payee, this);

}

#endregion

The example code below was generated using the polycoll snippet:

#region PayableItems Collection of type IPayableItem

private ICollection<Payment\_PayableItem\_Link> \_PayableItem =   
 new List<Payment\_PayableItem\_Link>();

[NakedObjectsIgnore]

public virtual ICollection<Payment\_PayableItem\_Link> PayableItemLinks {

get {

return \_PayableItem;

}

set {

\_PayableItem = value;

}

}

public void AddToPayableItems(IPayableItem value) {

PolymorphicNavigator.AddLink  
 <Payment\_PayableItem\_Link, IPayableItem, PolymorphicPayment>(value, this);

}

public void RemoveFromPayableItems(IPayableItem value) {

PolymorphicNavigator.RemoveLink  
 <Payment\_PayableItem\_Link, IPayableItem, PolymorphicPayment>(value, this);

}

[NotPersisted]

public ICollection<IPayableItem> PayableItems {

get {

return PayableItemLinks.Select(x => x.AssociatedRoleObject).ToList();

}

}

#endregion

##### Adding optional (database) referential integrity to a polymorphic association

As may readily be observed from the diagram above, at this stage the database schema does not have referential integrity across the PA; and navigating for reporting purposes, while possible, will not be easy. If this is an issue for your application, then it may optionally be addressed added by manually adding further tables as shown in the example below:



Notice that for each of the polymorphic link tables (on the right hand side) there is now an added table *for each of the implementations of that role.* Thus PayeeLinks has associated PayeeLinks\_Customer and PayeeLinks\_Employer tables. Each of these ‘second half’ tables (on the left-hand side of the diagram) has an FK both to a row in the corresponding Link table, and to its ‘role’ object. These ‘second-half’ tables will need to be created manually, and maintained by custom database triggers: they are invisible to Entity Framework and the domain code.

## Customising the Restful API

#### How to determine whether an action will require a GET, PUT or POST method

Restful Objects for .NET adopts the convention that any action that returns an IQueryable of a valid domain object type will be treated as a 'query only' action (side-effect free) method, and may therefore be invoked using the http GET method. (The programmer must therefore ensure that any actions that change the state of the system do not return an IQueryable result).

All other actions will, by default, require a POST method.

However, this default behaviour may be over-ridden by annotating the method (in the domain model) with a QueryOnly attribute (which will allow invocation via a GET) or Idempotent attribute (invocation via PUT). The application developer must take care to ensure that these attributes are applied correctly: they will impact the way that the action is rendered via the restful interface, but do not enforce that the method's behaviour is consistent with the attribute.

For further information about these distinctions, please refer to the Restful Objects specification.

#### How to work with proto-persistent objects

The Naked Objects Restful API supports proto-persistent objects as defined in the Restful Objects spec - but these are referred to as 'transient objects' within the Naked Objects programming model. When persisting a transient object (via a POST to the objects resource), you will need to provide all of the properties of the object - including any that might be disabled or hidden. For example, a method on the Customer object might return a new transient instance of an Order, having set the Customer property on the Order in order to associate it with the Customer when it is persisted. But this Order.Customer property will almost certainly be disabled and might even be hidden, since the user typically does not need to be reminded of it.

This is usually straightforward to handle, since the values of any properties that were set up programmatically before the transient object was returned from the server will be included in the arguments map associated with the persist link.

Where it becomes slightly tricky is if the transient object has Disabled or Hidden properties that are not always populated. (This is, admittedly, a fairly obscure use-case). In this circumstance the Restful Objects server may return an error, indicating that values have not been supplied for mandatory properties. Normally, a property that is Disabled or Hidden would be ignored when checking for mandatory properties, but in the case of transient objects it is not. If this situation arises, then the solution is to also mark those properties explicitly with the Optionally attribute.

#### How to enforce concurrency-checking

To switch on concurrency checking, add add the following line of code into the RegisterRoutes method on the NakedObjectsStart class in your run project.

RestfulObjectsController.ConcurrencyChecking = true;

With this capability switched on, the client will need to provide the if-match header information, as defined in the Restful Objects specification. And if your application is using the Entity Framework for persistence, you will need to use the [ConcurrencyCheck](#_ConcurrencyCheck) attribute.

In order for the RESTful API generated by the server to be compliant with the Restful Objects specification, concurrency checking must be switched on. The only reason that it has been made switchable in the implementation, is because it can be burdonsome in the early stages of application development, because clients must provide the If-Match header.

#### How to make an application Read Only

If your application is intended as read-only, add the following line of code into the RegisterRoutes method on the RestfulObjectsStart class in your run project.

RestfulObjectsController.IsReadOnly = true;

The effect of this will be to disallow all but GET methods. Links to any PUT, POST or DELETE methods may still be shown, but any attempt to invoke them will result in an error.

#### How to implement automatic redirection for specific objects

If a domain object implements the interface NakedObjects.Redirect.IRedirectedObject (defined in the NakedObjects.Types.dll) then when an attempt is made to retrieve the object (either directly via a link to the object resource, or as an action result), the server will return an http '301 Re-direct' to an alternative Restful Objects object resource that may even be on another server.

This pattern is particularly effective as a mechanism for integrating multiple existing systems. In the following example, the Order object may be thought of as existing in one system (‘A’) and the Customer in on another system (‘B’). However, system A has a very light-weight ‘stub class’ called CustomerMemento, which has minimal content, but implements IRedirectedObject.

The Customer class on the system B is in fact merely a 'stub' - the class is persisted on the new system (which is why it has its own Id property), but the persisted values for ServerName and Oid provide the necessary information for constructing a Url to the full Customer object on the system B. The Title property contains information that is duplicated from the data in the old system - to reduce the need for the user to navigate the link in order to see the identity of the customer – this is not required to implement IRedirectedObject. (Ideally, this title will be static identity information: if it were subject to change then it would be necessary to add a mechanism to enforce consistency).

//Stub object

public class Customer : IRedirectedObject {

public virtual int Id { get; set; }

[Title]

public virtual string Title { get; set; }

public virtual string ServerName { get; set; }

public virtual string Oid { get; set; }  
}

#### How to change cache settings

Based on the three 'shorthand' types of response defined in the Restful Objects specification, the default cache settings for the Restful Objects for .NET implementation are as follows,

* Transactional representations: 0 seconds (i.e. no caching)
* User representations: 3600 seconds (1 hour)
* Non-expiring representations: 86400 seconds (1 day)

These may be over-ridden by adding the following line into the RestfulObjectsStart#RegisterRoutes method, and changing the three integer values as required:

RestfulObjectsControllerBase.CacheSettings = new Tuple<int, int, int>(0, 3600, 86400);

#### How to handle Complex Types

Restful Objects for .NET does not currently support Complex Types. If a domain object does have a property that is a Complex Type, then this will be rendered the same way as a regular reference property (i.e. an association), complete with the title of the complex type object. However, the url in the link just points to the parent object (i.e. the object you are currently viewing). There is currently no way to view or edit the properties of the complex type object directly.

If you absolutely need to use a Complex Type and you need to view and/or edit the properties of that complex type (other than just viewing the title) then you will need to add one or more dedicated actions, as illustrated with the following example.

public class Customer {

//This property is a complex type, see below

[Disabled]

public Address HomeAddress { get; set; }

public void EditAddress(string line1, string line2) {

HomeAddress.Line1 = line1;

HomeAddress.Line2 = line2;

}

public string Default0EditAddress() {

return HomeAddress.Line1;

}

public string Default1EditAddress() {

return HomeAddress.Line2;

}

}

[Inline] //Complex Type

public class Address {

//Title (would probably truncate each line)

public string ToString() {

return Line1 + Line2;

}

public string Line1 { get; set; }

public string Line2 { get; set; }

//etc

}

#### How to change the format of the Object Identifier (Oid) in resource URLs

Many of the resources (URLs) defined in the Restful Objects specification include an 'object identifier' in the form: {DType}/{IID}, where {DType}is the domain type identifier and {IID} is the instance identifier; taken together these are referred to as the object identifier or Oid. By default, the Restful Objects for .NET server renders the domain type identifier as the fully-qualified class-name, and the instance identifier as the key of the object or, if it has a compound key, as the keys separated by dashes e.g.:

MyApp.Customers.WholesaleCustomer/10876

MyApp.Sales.OrderLine/8566--1055

You can override the format of either or both parts of the Oid. This might be for readability, for example:

WholesaleCustomer/10876

ORDLIN/8566--1055

or it might be to encrypt one or both parts, to prevent a rogue user from guessing the URL of an object they could not otherwise retrieve (though note that this could also be prevented through the Naked Objects custom authorization):

MyApp.Customers.WholesaleCustomer/f56bk-xx803h-jk4788ggweq2

Yet another reason would be if a domain object has one or more string keys that contain the default key separator. For example, if you have a natural (single) key of sku-10452, then the Restful Objects Server would, by default, interpret that as a two-part key and be unable to retrieve the object.

Control over the format is managed by creating an implementation of one or both of the following service definitions, each of which defines just two simple methods for converting each way:

* NakedObjects.ITypeCodeMapper - to take control over the format of the domain type identifier.
* NakedObjects.IKeyCodeMapper - to take control over the format of the instance identifier.

Your implementation(s) of these service interfaces should then be [registered](#_Registering_domain_services).

#### How to limit the scope of the domain model that is visible through the Restful API

By default, Restful Objects for .NET will create a Restful API to the whole of your domain model. (Strictly speaking, it is to all the parts of your domain model that can be reached directly, or indirectly, from the run project). If you wish to restrict the Restful API to a sub-section of the domain model, then there are several different techniques available to you:

* Use custom authorization to control visibility of classes, properties and actions to certain users or groups of users. Note that this approach could also be used to hide sections of the model from all users - without necessarily impacting any user interface - because the two 'interfaces' can be generated by two different run projects, each of which can register a different custom authorizer.
* Make use of the [ITypeCodeMapper](#_NakedObjectsServices). The primary intent of these interfaces was to allow control over the format of the object -identifier in resource URLs. Note, however, that they could also be used simply to ensure that large sections of the domain model, or possibly certain ranges of object Ids, can never be mapped as resource URLs in the Restful API. For example, the following code would ensure that nothing in the MyApp.Payments or MyApp.Employees namespaces of the domain model is mapped:

public class CustomRestrictiveTypeCodeMapper : ITypeCodeMapper {

public Type TypeFromCode(string code) {

return IsExcluded(code) ? null : TypeUtils.GetType(code);

}

public string CodeFromType(Type type) {

string typeName = TypeUtils.GetProxiedTypeFullName(type);

return IsExcluded(typeName) ? null : typeName;

}

private bool IsExcluded(string typeName) {

return typeName.ToUpper().StartsWith("MYAPP.PAYMENTS") || typeName.ToUpper().StartsWith("MYAPP.EMPLOYEES");

}

}

The use of the two methods TypeUtils.GetType and TypeUtils.GetProxiedTypeFullName. These are the safest ways to convert between Types and strings in Naked Objects. Calling foo.GetType()on a persistent object will return the Entity Framework proxied type, not the raw domain type - using TypeUtils.GetProxiedTypeFullName will get you the raw domain type.

Returning null from each of the methods when the intended type is to be excluded is sufficient to ensure that that type can never be accessed via the Restful API. A similar approach could be used with an implementation of IKeyCodeMapper to exclude certain ranges of instance Ids if desired.

#### How to switch off strict Accept header enforcement

By default, the Restful Objects for .NET server strictly enforces Accept headers. Unless the request includes, at minimum, application/json (or a broader match such as \*/\*) then a 406 error will be returned.

Sometimes, during the early stages of development this can be a little inconvenient, especially if you are using a client tool that does not allow Accept headers to be set up by default. In these circumstances it is possible to switch off strict Accept header enforcement, by adding the following code into the RestPreStart method on the RestfulObjectsConfig class:

RestfulObjectsControllerBase.AcceptHeaderStrict = false;

You should therefore ensure that strict enforcement is switched back on again before you deploy the live application.

#### How to limit the size of returned collections

The Restful Objects 1.0 specification does not provide explicit support for paging through returned collections. A sketch of a possible approach to paging is included in Section 34.3 of the 1.0 specification under 'Ideas for Extensions to the Specification' - and something like this is likely to be added to a future release.

Currently the Restful Objects for .NET server will only return the first 'page' (meaning the first 20 elements) from any collection. This is to avoid the risk of accidentally returning thousands of elements. If the returned collection has more elements, only the first 20 will be shown, but the returned header will contain a warning message (code 299) that this has occurred.

You can override the default page size of 20 in the RegisterRoutes method on the RestfulObjectsStart class in your run project, for example:

RestfulObjectsControllerBase.DefaultPageSize = 50;

A value of 0 means 'unlimited page size' - query methods will return all elements in the collection. This should be used with caution.

Pending the introduction of framework-level support for paging, you can implement paging within your own query methods, for example:

public IQueryable<Order> Orders(int pageNumber){

int pageSize = 50;

int skip = pageSize\*pageNumber;

return Container.Instances<Order>().Skip(skip).Take(pageSize);

}

Note that for the above code to work, you should ensure that the DefaultPageSize is at least as large as the page size used within your query methods.

#### How to enable cross-origin resource sharing (CORS)

If your Restful Objects client is browser-based and is not loaded from the same location as the Restful Objects Server, then you are probably going to run into the 'same origin policy' issue. The easiest way to proceed is to enable cross-origin resource sharing (CORS). CORS can also be useful if you want to create a client that will integrate functionality from multiple Restful Objects servers.

You can implement CORS for yourself, but a much easier solution is to use the Thinktecture.IdentityModel NuGet package.

1. Add Thinktecture.IdentityModel to the run project through the NuGet Package Manager.
2. Add a CorsConfig.cs file under App\_Start:

using System.Web.Http;

using Thinktecture.IdentityModel.Http.Cors.WebApi;

public class CorsConfig {

public static void RegisterCors(HttpConfiguration httpConfig) {

var corsConfig = new WebApiCorsConfiguration();

// this adds the CorsMessageHandler to the HttpConfiguration's

// MessageHandlers collection

corsConfig.RegisterGlobal(httpConfig);

// this allows all CORS requests to the RestfulObjects controller

// from the http://foo.com origin.

corsConfig.ForResources("RestfulObjects").ForOrigins("http://foo.com").AllowAll();

}

}

1. Hook in the config in RestfulObjectsStart.PostStart

public static void PostStart() {

// existing code

.......

// add CORS

CorsConfig.RegisterCors(GlobalConfiguration.Configuration);

}

#### Authorization in Restful Objects

The Restful Objects Server honours the same approaches to authorization as Naked Objects MVC - including attribute authorization and, custom authorization. However, you will need to provide your own mechanism for determining the user and roles based on the approach to authentication that you wish to use on your application.

By way of illustration only, the Restful Objects Server package installs a BasicAuthenticationHandler (in the App\_Start folder). In order to use this, you must register it within the PostStart() method in RestfulObjectsStart (also to be found in the App\_Start folder), as shown here:

public static void PostStart() {

...

GlobalConfiguration.Configuration.MessageHandlers.Add(new BasicAuthenticationHandler());

}

This BasicAuthenticationHandler checks the incoming request to see if it has any credentials. If not, it raises a simple LogIn request back to the client.

The BasicAuthenticationHandler also contains the following method:

private UserCredentials Validate(string user, string password) {

//THIS CODE IS A MOCK IMPLEMENTATION FOR ILLUSTRATION ONLY

//Fail if either the user name or password is null or empty

if (string.IsNullOrWhiteSpace(user)) return null;

if (string.IsNullOrEmpty(password)) return null;

//Otherwise, authenticate any user/password combination, with a standard set of roles

return new UserCredentials(user, password, new List<string> { "Role1", "Role2" });

}

As the comments make clear, this basic code does not validate the user name or password; a real implementation would validate these against a database or, more likely, delegate to some authorization service. The last line of the code associates two default roles with the user - again this could be delegated to an external service.

Meantime, for exploration and testing purposes only, you can modify the last line (or any of the code in this method) to ensure that you are working with a user that has the correct roles assigned.

#### How to hook additional functionality into controller methods

Create a new ‘filter’as a sub-class of the ActionFilterAttribute:

public class MyActionFilter : ActionFilterAttribute {

public override void OnActionExecuting(HttpActionContext actionContext) {

//....my new functionality here

            var i = Guid.NewGuid();

        }

        public override void OnActionExecuted(HttpActionExecutedContext actionExecutedContext) {

            //....and here

            var i = Guid.NewGuid();

        }

    }

}

Then add attribute to any methods on RestfulObjectsController that you require to extend e.g:

 [HttpGet]

[MyActionFilter]

 public override HttpResponseMessage GetHome([ModelBinder(typeof   
 (ReservedArgumentsBinder))] ReservedArguments arguments) {

return base.GetHome(arguments);

}

the HttpActionContext gives access to the Request and to the Controller if required.

#### How to inject services or the Domain Object Container into the controller:

Use the Inject method on the IFrameworkFacade, which is passed into the constructor of the controller:

public IDomainObjectContainer Container { protected get; set; }

public RestfulObjectsController(IFrameworkFacade frameworkFacade) : base(frameworkFacade) {

frameworkFacade.Inject(this);

}

## Adding custom views to the UI

NOF8 does allow you to create customised views for particular types of object, to replace the generic object views. *However, you should do so with extreme caution:* adding lots of custom views can create a better user experience, but this is almost always at the expense of the maintainability of the application. Furthermore, each time you upgrade the Naked Objects packages you will have to take responsibility to upgrade your custom views and/or controllers.

Apart from simple customisations, such as changing the layout, you will need to have a good working knowledge of the [AngularJS](https://angularjs.org/) framework, and be able to figure out for yourself how this is being used with the various NOF script files. Note that these NOF script files are all written in the [TypeScript](https://www.typescriptlang.org/) language (the .ts files) and compiled down to JavaScript (.js).

For simple customisations, however, the following will get you started. Let’s say that we want to have a customised view of a Product object. To do this we will need to create a view template and give it a name such as productView.html. (This naming is purely a convention, but it is useful in case we also want to have a customised ‘edit’ representation for the same type of object, which we might call productEdit.html).

As a start-point you can create as productView.html by copying the generic template: objectView.html and renaming it. As a first step you can make a trivial modification to the view to prove that it is being used whenever a Product object is displayed. For example we might change this generic line:

<div class="title" gemini-drag tabindex="0">**{{**object.title**}}**</div>

to:

<div class="title" gemini-drag tabindex="0">My custom view of: **{{**object.title**}}**</div>

We can place the new file in the same folder (Content > Partials), or in a new folder, Content > CustomViews, say.

The final step is to associate the new view with the Product object type by adding a line into the existing nakedobjects.services.template.config.ts file, for example:

module NakedObjects {

app.run((template: ITemplate) => {

template.setTemplateName("AdventureWorksModel.Product",   
 TemplateType.Object, InteractionMode.View,   
 "Content/customTemplates/productView.html");  
  
 template.setTemplateName("AdventureWorksModel.Product",

TemplateType.Object, InteractionMode.Edit,   
 "Content/customTemplates/productEdit.html");  
  
 template.setTemplateName("AdventureWorksModel.Product",   
 TemplateType.List, CollectionViewState.Table,   
 "Content/customTemplates/productTable.html");

});

}

In the example code above, we have specified three custom templates for viewing a Product object, editing a Product object, and viewing a returned list of Product objects as table. It is *not* necessary to customise all the templates for rendering a given type: you can specify any one of the custom templates, leaving other modes to use the default templates.

The following are some specific custom view patterns that you can experiment with.

### Layout of properties

By default, all object properties are rendered according to their Member Order as defined in the domain code by incorporating the properties.html template within the objectView.html, thus:

<div class="properties" ng-include="'Content/partials/properties.html'"></div>

You can replace this with your own template, for example:

<div class="properties" ng-include="'Content/CustomViews/groupedProps.html'"></div>

where groupedProps.html specifies where to render each property individually, with headings between them, for example:

<h2>Details</h2>

<div class="property" ng-repeat="property **in** object.properties">

<div ng-if="property.title == 'Name'"   
 ng-include="'Content/partials/readOnlyProperty.html'"></div>

</div>

<h2>Dates</h2>

<div class="property" ng-repeat="property **in** object.properties">

<div ng-if="property.title == 'Start Date'"  
 ng-include="'Content/partials/readOnlyProperty.html'"></div>

</div>

<div class="property" ng-repeat="property **in** object.properties">

<div ng-if="property.title == 'End Date'"  
 ng-include="'Content/partials/readOnlyProperty.html'"></div>

</div>

Note that we are still delegating to the standard lower-level template readOnlyProperty.html for rendering each property.

A more sophisticated custom view might group the properties using an ‘accordian’ component, such that the groups could be expanded/collapsed by the user, as needed.

### Render an action as a top-level button

By default, all object actions are rendered within a menu, opened when the user clicks on the Actions button. Alongside Actions there may be other generic buttons such as Reload and Edit. If you had a frequently-used action that you would like rendered as a top-level button, then you could do this by adding this code into the custom view within the header div:

<div class="header">

...

<!--After the Actions, Edit and Reload buttons-->

<div ng-repeat="action **in** object.actions">

<div ng-if="action.title == 'Create New Order'"   
 class="action clickable-area"   
 title="**{{**action.description**}}**"   
 tabindex="0" ng-click="action.doInvoke()"   
 gemini-enter="action.doInvoke()"   
 gemini-rightclick="action.doInvoke(true)"   
 ng-disabled="action.disabled()">

**{{**action.title**}}**

</div>

</div>

...

</div>

Note that, with the exception of the ng-if clause (to select just the Create New Order action), the code to render the action is the same as found in the standard actions.html template that renders the actions within the Actions menu.

### Polling for updates

We might want a view of a particular object (typically a domain [view model](#view_model)) to poll regularly for updates. The simplest way to do this is to reload the whole object at period intervals. This can be achieved by means of a custom (Angular) controller. For example, the custom view might begin:

<div ng-controller="PollingController" class="object view" ng-class="object.color">

<div class="header">

A simple implementation of PollingController might look like this:

app.controller("PollingController", ($scope: INakedObjectsScope, $timeout: ng.ITimeoutService, urlManager: IUrlManager) => {

const scope = $scope;

const reload = () => $timeout(() => {

// keep reloading while on page

const rd = () => urlManager.getRouteData().pane()[scope.object.onPaneId];

if (ObjectIdWrapper.fromObjectId(rd().objectId)  
 .isSame(scope.object.domainObject.getOid())) {

scope.object.doReload();

reload();

}

}, 10000);

reload();

});

Note: If you write this controller in a new script file, you will need to include it (the JavaScript version of it, that is) in the Index.html page along with the existing scripts.

## Other

### Getting hold of the current user programmatically

If your domain model needs to get direct access to the user name of the current user, this may be obtained via the Principal method on the injected Container. The Principal method returns a System.Security.Principal.IPrincipal, from which you may obtain the user's identity and thence the name. This may then be used for example to retrieve a domain object such as an Employee that represents that user. For example, the following method returns the Employee object representing the logged on user:

public class EmployeeRepository {

//(Add Injected Container)

public Employee CurrentUser() {

var userName = Container.Principal.Identity.Name;

var query = from employee in Container.Instances<Employee>()

where employee.UserName == userName

select employee;

return query.FirstOrDefault();

}

}

**Recommended pattern for accessing the current user as a domain object.**The recommended pattern is to create an interface that defines a single templated method as follows:

public interface IUserService {

T CurrentUser<T>();

}

And then to implement this interface in the repository for whichever type of domain object also corresponds to users - for example:

public class EmployeeRepository : IUserService {

public T CurrentUser<T>() {

if (! (typeof(T).IsAssignableFrom(typeof(Employee)))) {

throw new DomainException("Cannot convert an Employee to type:" + typeof(T).ToString());

}

var userName = Container.Principal.Identity.Name;

var query = from employee in Container.Instances<Employee>()

where employee.UserName == userName

select employee;

return query.Cast<T>().FirstOrDefault();

}

}

This allows other objects to have the IUserService injected, and to call CurrentUser() with a type T (for example: ICommunicablePerson) that they are interested in, without being coupled to Employee - provided that Employee implements or extends that type.

### Creating an XML Snapshot of an object

Sometimes it can be very useful to create an XML Snapshot of a domain object, for example:

* For auditing purposes (to capture and store the complete state of an object at a given moment).
* To facilitate merging data from the model into a letter-template
* To exchange information with an external service that uses XML

To use this capability, you will need to [register](#_Registering_services) the NakedObjects.Snapshot.XMLSnapshotService.

This service can be injected into any domain object with a suitable property of type NakedObjects.Snapshot.IXmlSnapshotService, an interface which is defined in NakedObjects.Helpers.dll (installed into your model project as part of the NakedObjects.ProgrammingModel package). You can then call the GenerateSnapshot method on this service, passing in the domain object of interest. Having generated the snapshot, but before reading the XML itself (via its Xml property), you can also specify associated objects that should be in-lined within the snapshot - navigating as far down the graph of associated objects as you wish. All these ideas are illustrated in the code below, but we recommend that the best way to learn how this works is simply to experiment:

public class Product {

...

//Injected service

public ProductRepository Snapshotter {set; protected get}

public void Archive() {

IXmlSnapshot = Snapshotter.GenerateSnapshot(this);

snap.Include("Manufacturer"); //In-lines an XML representation of the object in the Product's Manufacturer property

snap.Include("Manufacturer/Address"); //In-lines an XML representation of the object in the Manufacturer's Address property

string xml = snap.Xml;

...

}

}

IXmlSnaphot also provides methods to return the XML schema definition (Xsd), and to transform the generated XML using a standard XSL Transform passed in as a string (TransformedXml).

## How to run Naked Objects as a .exe

Follow these steps

1. Add into your solution a new 'Windows Application' project. The simplest way to do this is to add a project of type 'Console Application' and then in the project's Application Properties, set the Output Type to Windows Application (as you don't typically want the console to appear).
2. Delete any existing Program.cs file, so that it can be replaced by the next step ...
3. Using the NuGet Package Manager, add the NakedObjects.Batch package . This will add a new Program.cs file:

public class Program {

public static void Main() {

UnityActivator.Start();

UnityConfig.GetConfiguredContainer().Resolve<IBatchRunner>().Run(new BatchStartPoint());

UnityActivator.Shutdown();

}

}

1. [Configure the application](#fixtures) in the NakedObjectsRunSettings class.
2. The package install will also have added a class BatchStartPoint; add your code into the (empty) Execute method. Alternatively, you may provide any class that implements NakedObjects.Boot.IBatchStartPoint, and instantiate that class within the Run method, above. Note that either BatchStartPoint, or your own implementation of IBatchStartPoint can have the IDomainObjectContainer and/or any registered domain services injected into it - to give you access to the domain model from this start point.

You may choose to leave this .exe running permanently on your server, or to launch it at regular intervals via the Windows Task Manager.

If your batch implementation can be broken down in a series of separate tasks, then, rather than performing them sequentially, you should consider using the Naked Objects Async capability...

## How to run multiple threads asynchronously

Naked Objects provides a convenient mechanism for running multiple Naked Objects threads in parallel. You will need to inject a NakedObjects.Async.IAsynchService (defined within the NakedObjects.ProgrammingModel package) into your code:

public IAsyncService AsyncService { set; protected get; }

(The Naked Objects framework provides an implementation of IAsyncService NakedObjects.Async.AsyncService. You will need to [register this service](#_Registering_domain_services), or another implementation if you wish to write your own).

You can call the RunAsync method on this service, passing in the action to be run as a lambda. This method returns a System.Threading.Tasks.Task; by collecting these tasks into an array you can then instruct the system to wait until all such tasks are completed, using the standard .NET code of Task.WaitAll(tasks), as illustrated in the following example code:

public ProcessRepository ProcessRepository { set; protected get; }

public AsyncService AsyncService { set; protected get; }

public void RunAllProcessesDueBy(DateTime dateTime) {

IList<ProcessDefinition> due =   
 ProcessRepository.GetProcessesWithNextRunDueBy(dateTime);

var tasks = due.Select(pd => AsyncService.RunAsync((domainObjectContainer) =>   
 FindAndRunProcess(pd.Id))).ToArray();

Task.WaitAll(tasks);

}

private void FindAndRunProcess(IDomainObjectContainer container, int processId) {

var repository = container.GetService<ProcessRepository>();

var proc = repository.GetProcess(processId);

proc.Run();

}

In the example above, the ProcessRepository and ProcessDefinition are straightforward domain classes. The first line just obtains a set of objects that contain details of processes due to be run by the specified DateTime, each with a Run() method. The second line invokes a method on to find and run each of those due processes, asynchronously. The third line waits until all the asynchronous processes are completed.

Each of the asynchronously-run processes is fully independent. You should therefore not pass domain objects into or between these processes, but you may pass .NET primitives, including object Ids - to be retrieved independently inside the new thread. This is why, in the above example, FindAndRunProcess is passed in the Id of the ProcessDefinition, not the domain object itself, nor the ProcessRepository. The method RunAsync can pass in a properly set-up instance of IDomainObjectContainer into the method to be run, and this can then be used to obtain domain service(s) and, thence, domain instances. **Note**: The Naked Objects AsyncService is designed only to be used within a standalone executable run project (as illustrated above). It will not work within an HttpContext.

## Authorization

Authorization, means the ability to control what an individual user can see and do within an application, based upon their identity, the role(s) assigned to them, and/or other credentials or 'claims'. (It follows that authentication is a necessary precursor to authorization.) Naked Objects supports 'fine-grained authorization', meaning that it is possible to specify whether or not a user may View - and, separately, whether they may Edit - individual properties on object types, and whether or not they may invoke individual actions on objects or services.

Currently, there are three choices for authorization:

* **Attribute-based Authorization**. This is the default mechanism for authorization. Simply mark up any properties or action methods with attributes that specify the Roles (or users) to whom they should be available. This approach is best suited to applications where you have a relatively simple domain model and/or a small number of distinct user roles. Attribute-based authorization may be used with any of the three approaches to authentication listed above.
* **Custom authorization**. In this approach you plug-in your own mechanisms for managing authorization, which may be self-contained or may delegate responsibility to an external service. Using this approach it is possible to control authorization at the level of individual object instances.

### Attribute-based Authorization

In this approach, authorization rules are implemented by applying attributes to properties and/or actions inside the domain model classes. If any property or action does not have an authorization attribute specified then that property/action is assumed to be available to all users.

Consider the following example:

Imports NakedObjects.Security

Public Class Customer

[AuthorizeAction(Users:="fred", Roles:="CustomerService, Manager")>

Public Function PlaceNewOrder() As Order

...

End Function

...

End Class

using NakedObjects.Security;

public class Customer {

[AuthorizeAction(Users="fred", Roles="CustomerService, Manager")]

public Order PlaceNewOrder() {...}

...

}

Here the user action Place New Order will only appear on the menu of users who have either the role CustomerService, or Manager, plus the user named Fred who will see the action irrespective of his role(s).

Both the syntax and the semantics of NakedObjects.Security.AuthorizeAction attribute are similar to those of the System.Web.Mvc.Authorize attribute - so why not just use the latter? In part this is because we do not want domain models to become dependent upon ASP.NET MVC - or any other specific user interface (the NakedObjects.Security.AuthorizeAction is contained within the tiny NakedObjects.Attributes assembly with the other Naked Objects attributes). The other reason is that the System.Web.Mvc.Authorize attribute would not allow the separate specification of View and Edit rights for properties, as described below.

For properties, we use the NakedObjects.Security.AuthorizeProperty attribute. This allows us to specify roles and/or users that may View the property, and, separately, roles and/or users that may Edit the property. Consider the following example:

<AuthorizeProperty(ViewRoles:="CustomerService, Finance", EditRoles:="Finance")>

Public Overridable Property CreditRating() As Decimal

[AuthorizeProperty(ViewRoles="CustomerService, Finance", EditRoles="Finance")]

public virtual decimal CreditRating {get; set;}

Here, the Credit Rating property may be viewed by any user with the role CustomerService or Finance, but may only be edited by a user with the role Finance. (Individual users may also be authorized, by adding ViewUsers and EditUsers parameters to this attribute).

A user or role will not be able to Edit a property unless they are also authorized to View that property - hence the appearance of Finance in both the ViewRoles and EditRoles above.

The AuthorizeProperty and/or the AuthorizeAction attribute, may also be applied at class level:

<AuthorizeAction(Roles:="Manager")>

<AuthorizeProperty(ViewRoles:="CustomerService, Manager", EditRoles:="CustomerService, Manager")>

Public Class Customer

...

End Class

[AuthorizeAction(Roles="Manager")]

[AuthorizeProperty(ViewRoles="CustomerService, Manager", EditRoles="CustomerService, Manager")]

public class Customer {...}

This then applies the authorization rules to all properties and/or actions within that class. The class-level attributes will over-ride any such attributes applied to individual properties or actions within that class, including properties/actions specified on sub-classes. Attempting to mix the two approaches is not recommended. On initialisation of the system, if such an over-ride is detected, a system warning will be raised.

### Custom Authorization

Custom authorization is specified using one or more authorizer classes, each which implements *either* ITypeAuthorizer<T>, where T is a concrete type from your domain model, or INamespaceAuthorizer (to provide authorization logic for any types within a specified namespace).

Each authorizer class must be [registered](#_Configuring_Authorization).

When the framework needs to determine authorization for a given object, it will search for the authorizer that provides the most specific match to that type object; if no such authorizer is found, it will use the default authorizer, which by implementing ITypeAuthorizer<object> will be able to work with domain objects of all types. Thus, you will typically only add a type authorizer for types (or groups of types within a namespace) that require their own specific approach to authorization.

The code below shows a skeletal implementation of the FooAuthorizer:

public class FooAuthorizer : ITypeAuthorizer<Foo> {

public bool IsEditable(IPrincipal principal, Foo target, string memberName) {...}

public bool IsVisible(IPrincipal principal, Foo target, string memberName) {...}

}

* IsVisible is called to determine whether a given object-member (a property or an action method) should be visible to the current user. The user-name may be derived from the principal parameter; the specific object instance is passed in as the target parameter (this is needed only if you are writing 'instance-based authorization' logic); the third parameter gives the memberName.
* IsEditable is called to determine whether a given object property may be edited by the user. IsEditable has no determined meaning for action members.

Within the IsVisible and IsEditable methods, you can either write custom logic, or delegate out to some external service.

There is a useful method in TypeUtils (within the NakedObjects.Helpers assembly) that facilitates type-safe testing of a property name, for example:

TypeUtils.IsPropertyMatch<Customer, dateTime>(target, memberName, x => x.DateOfBirth).

This is especially useful within an implementation of INamespaceAuthorizer as the method checks both the type of the target and the match for the memberName.

An implementation of ITypeAuthorizer<T> or INamespaceAuthorizer can have domain services and/or the IDomainObjectContainer injected into it.

#### Controlling visibility of columns in a table view

In a table view the authorizer will be called for each object instance being rendered in the table. If a given property is not visible for any of the instances (rows) in the table then that property will not be rendered as a column in the table. If the property is visible for any instance(s) being rendered then the column will be shown, but the value within it rendered only for those instances that authorize its visibility.

### Adding further security measures

From a security perspective, Naked Objects offers the huge advantage that authorization is enforced both at the user interface and at the server API. If a given user is not authorized to invoke a particular action, then that action will not be rendered on the default user interface. Moreover, if a user deliberately sought to by-pass this limitation by interacting directly with the Restful API of the server, constructing the URL and any necessary JSON arguments by hand, then the attempt would fail. The same would apply to attempting to update a property of an object that the user was not authorised to modify.

One potential hole in this protection is in the persisting of a new object. In normal usage, the user invokes an action (typically called something like Create new…) that returns a transient object as a form for the user to complete and then hit the Save button. Unauthorised users will not be presented with the Create new… action. However, an unscrupulous user could mock up the necessary JSON representation of a transient objects, and invoke the persist resource, without needing access to the Create new… action at all. Alternatively, they could alter fields on a transient object that were not intended to be modified (or perhaps not even seen) via the user interface.

To prevent this, when the server creates the transient object it creates a list of all of the properties that are not intended to be modified by the user (whether because they are read-only or hidden), adds the identity of the user, and creates a hash of them. This hash is passed up to the client with the data as an ETag. When the user saves an object the client must provide the hash value, and this is validated against the user identity and object properties.

The hashing is done by a component called SimpleStringHasher that is [registered in the unity.config file](l%20), thus:

container.RegisterType<IStringHasher, SimpleStringHasher>  
 (new PerResolveLifetimeManager());

SimpleStringHasher uses the SHA-256 algorithm but does not use salting or provide any encryption, so a determined (and technically-competant) hacker could still fake the hash value. If this is a concern for your application, then you may simply replace the default component with your own implementation of IStringHasher that includes salting and/or another form of encryption.

***Note****: There is a rare circumstance in which a transient object has a derived field that is dependent upon a field that the user enters, which can result the hashes not matching. If you should encounter this issue, simply mark the derived property with [NotPersisted].*

Alternativelty, if you do not need this level of security (because your application is for internal users that can be trusted not to attempt to hack the system), you can replace SimpleStringHasher with ConstantStringHasher – the latter returns a constant string that takes no account of the field values and hence will always match up irrespective of any changes.

## Auditing

Naked Objects provides a general purpose mechanism for recording user actions, either to support formal auditing, or just to provide a user-accessible mechanism for identify who did what. In order to use the audit capability, you need to create one or more ‘audit services’ and then [register](#_Configuring_Auditing) them.

Each time the user persists or updates an object, or invokes an action (on a service, or on a persisted object), the framework will look for the auditor that most precisely fits type of the object or service on which the action/update/persist has been invoked, and call the appropriate method, as defined on IAuditor:

void ActionInvoked(IPrincipal byPrincipal, string actionName, object onObject, bool queryOnly, object[] withParameters);

void ActionInvoked(IPrincipal byPrincipal, string actionName, string serviceName, bool queryOnly, object[] withParameters);

void ObjectUpdated(IPrincipal byPrincipal, object updatedObject);

void ObjectPersisted(IPrincipal byPrincipal, object persistedObject);

Note that the queryOnly parameter will be set to true if the action can be determined by the framework to be 'query only' i.e. any action that returns an IQueryable<T> and/or that is marked up with the QueryOnly attribute.

The implementations of IAuditor may manipulate the provided details of the user action and then optionally persist them as special ‘audit record’ domain objects. The persisted audit records may then be made available to suitably-authorised users via, say, an AuditRepository service.

Note that any implementation of IAuditor can have domain services and/or the IDomainObjectContainer injected into it.

## Profiling

If you want to profile your application, you can hook into various events generated by the framework, and then write out to a console, file, or logging framework. The pattern is somewhat similar to the patterns for [Authorization](#d0e8564) and [Auditing](#wif_authorization). Start by writing your own implementation of NakedObjects.Profile.IProfiler, for example:

public class MyProfiler : IProfiler {

public void Begin(IPrincipal principal, ProfileEvent profileEvent,  
 Type onType, string memberName) {

Debug.WriteLine(profileEvent.ToString() + " Start:" + DateTime.Now);

}

public void End(IPrincipal principal, ProfileEvent profileEvent,   
 Type onType, string memberName) {

Debug.WriteLine(profileEvent.ToString() + " End:" + DateTime.Now);

}

}

ProfileEvent is an enum that defines the set of Naked Objects events that can be profiled:

public enum ProfileEvent {

ActionInvocation,

PropertySet,

Created,

Deleted,

Deleting,

Loaded,

Loading,

Persisted,

Persisting,

Updated,

Updating,

}

Next you need to specify which of these events you wish to profile as a set, for example:

new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation, ProfileEvent.Updating, ProfileEvent.Updated };

Note that in addition to ActionInvocation, there are events that correspond to the Life Cycle methods for a domain object. Note: *you do not have to have any of those methods explicitly on your domain object - the events are generated whether or not you need to add behaviour into them via explicit methods*.

When profiling a Naked Objects application you would not normally use the PropertySet event, as this profiles only the time taken on domain code behaviour associated with an individual property set - not the total time to update the object. If you wish to profile the time taken from when a use hits Save on an object-edit view, then you can monitor from the start of Updating to the end of Updated.

Finally, you must [configure the profiling](#_Configuring_Profiling) in UnityConfig.

## Testing

### Executable Application Tests (XATs)

Executable Application Tests (XATs for short) allow you to test your application's functionality from the perspective of a user. XATs test complete user scenarios. In effect they execute a sequence of user actions, and they provide specific methods for testing what a user is allowed to see or do at any point within a scenario.

The XAT framework is fully compatible with the Microsoft test framework.

Create an XAT project as follows:

1. Create a new MS Test project.
2. Install the NakedObjects.Xat NuGet package into this project

#### Creating an XAT test class

The easiest way to create an XAT class like this is to use this template:

Add New Item > Visual C# Items > Naked Objects > XAT

which is installed when you install the [NakedObjects.Ide](#_Using_the_Naked) NuGet package into your solution.

[TestClass()]

public class ExampleXAT : AcceptanceTestCase {

[TestInitialize()]

public void Initalize() {

InitializeNakedObjectsFramework(this);

StartTest();

}

protected override Type[] Services {

get {

return new Type[] {

typeof(EmployeeRepository),

typeof(ClaimRepository),

typeof(RecordedActionRepository),

typeof(RecordActionService),

typeof(RecordedActionContributedActions),

typeof(DummyMailSender); }

}

}

#### Writing tests

The following code shows an example test. This method is testing functionality to create a new Expenses Claim in (an imaginary) Expenses Processing application:

[TestMethod()]

public virtual void ExampleTestMethod() {

...

SetUser("sven");

ITestObject claim = GetTestService("Claims").GetAction("Create New Claim").InvokeReturnObject("test");   
claim.GetPropertyByName("Description").AssertIsVisible().AssertIsModifiable();

claim.GetPropertyByName("Date Created").AssertIsVisible().AssertIsUnmodifiable();

...

claim.GetAction("Create New Expense Item").AssertIsVisible().AssertIsInvalidWithParms(null);

claim.GetAction("Copy An Existing Expense Item").AssertIsVisible().AssertIsInvalidWithParms(null);

claim.GetAction("Copy All Expense Items From Another Claim").AssertIsVisible();

...

claim.GetAction("Approve Items").AssertIsVisible();

...

claim.GetAction("Return To Claimant").AssertIsVisible().AssertIsInvalidWithParms("");

ITestParameter[] tps = claim.GetAction("Copy All Expense Items From Another Claim").Parameters;

tps[0].AssertIsMandatory().AssertIsNamed("Claim or Template");

tps[1].AssertIsOptional().AssertIsNamed("New date to apply to all items");

tps = claim.GetAction("Create New Claim From This").Parameters;

tps[0].AssertIsMandatory().AssertIsNamed("Description").AssertIsDescribedAs("");

tps[1].AssertIsOptional().AssertIsNamed("New date to apply to all items").AssertIsDescribedAs("");

ITestNaked[] choices = claim.GetAction("Create New Expense Item").Parameters[0].GetChoices();

Assert.IsTrue(choices.Length == 8, "Wrong number of choices in choice array");

((ITestObject)(choices[0])).AssertIsImmutable().AssertIsDescribedAs("");

claim.GetAction("Submit").AssertIsInvalidWithParms(null, true);  
}

Having specified that the test is to be run as though by the user named "sven" (because this particular application makes use of the user name to retrieve an Employee object corresponding to that name) the next line of the test may be read as follows:

On the service named *Claims*, invoke the action named *Create New Claim* with *test* as the single (string) parameter.

This action is expected to produce a (new) Claim object, which we store in a variable claim. But why is that variable specified as being of type ITestObject rather than of type Claim? This ITestObject can be thought of as a wrapper that holds the actual Claim object inside it. (Actually something very similar happens within the Naked Objects user interface, but this is transparent to both the user and the application programmer).

The ITestObject wrapper provides methods that allow you to get hold of the underlying objects properties and actions, for example:

claim.GetPropertyByName("Description")...

claim.GetAction("Create New Expense Item")...

which return an ITestProperty and an ITestAction respectively. These in turn provide methods for making assertions about the attributes or behaviour of those properties or actions from the perspective of the user, such as AssertIsVisible and AssertIsMandatory. For actions, you can drill down further and get to the individual parameters (as ITestParameter) for that action and make assertions about their attributes and behaviour.

##### Using the C# 6 nameof operator to create ‘refactor-safe’ XATs

If you are using C# 6, then it is possible to make use of the new nameof() operator, by means of new methods added (as of NOF 7.1). This means that your tests can be ‘refactor-safe’. For example:

claim.GetPropertyById(nameof(Claim.Description))...

claim.GetActionById(nameof(Claim.CreateNewExpenseItem))...

also:

GetTestService<ClaimRepository>()...  
laim.GetActionById(nameof(Claim.CreateNewExpenseItem))  
 .AssertHasFriendlyName("Create New Expense Item");

#### Simulating users and roles

The AcceptanceTestCase has a SetUser method that allows you to specify the name of the user and optionally the roles for the simulated user for the test. This allows you to test the visibility of actions and properties, and the modifiability of properties for a given user and/or roles, for example:

[TestMethod]

public void AuthorizedForEditAndView() {

SetUser("Bob");

prop1.AssertIsVisible();

prop1.AssertIsModifiable();

}

[TestMethod]

public void AccessByAnonUserWithViewRole() {

SetUser("Anon", "sysAdmin");

prop1.AssertIsVisible();

prop1.AssertIsUnmodifiable();

}

The form of authorization tested will depend upon how you have configured the test class. By default the test will assume use of attribute authorization. Or you may override the Authorizer property on the test class to specify, for example, custom authorization.

#### Running tests against a database

##### Resetting the database

The Naked Objects XAT framework provides a class NakedObjects.Xat.Database.DatabaseUtils, which provides methods for restoring the database to a known condition - typically a snapshot or a backup. (These utility methods make use of Microsoft's SQL Server Management Objects (SMO) framework.) You can make the snapshot/backup manually before running the tests, or create it automatically within the tests.

Backing-up or restoring a database can be a very time-expensive operation, so you should use this functionality with care. For example, it would be theoretically possible to restore the database for each individual test (within the Initialise method), which would guarantee that tests never trample on each other in the database. But in practice this is likely to be too time-consuming. A better approach would be to back-up/restore at class level (in a method marked up with the ClassInitialize attribute). In this case the programmer must make sure that the various test methods in that class do not trample on each other. This is only an issue when writing data, and can be simply managed by techniques such as:

* Ensuring that each test writes data to a different context - such as a different Customer.
* (Where this is not possible) Ensuring that any created objects are unique - for example by using a GUID as part or all of the content.

#### Using object fixtures within XATs

Object fixtures are used to set up objects for use in XATs, as an alternative to data fixtures (which are installed and managed by the database). The principal advantage of using object fixtures is that you can make use of methods and behaviour in the object model to set them up.

Naked Objects provides a dedicated pattern for creating and installing object fixtures before tests are run. In this pattern, an object fixture may be defined by any class that has a  
public void Install() method. Fixtures may have services (incluing the Container) injected into them, just like any domain object or service. The following code illustrates a simple object fixture class:

public class CustomerFixture {

public IDomainObjectContainer Container {protected get; set;}

public void Install() {

CreateNewCustomer("Fred Smith", "0001");

CreateNewCustomer("Joe Bloggs", "0002");

}

public Customer CreateNewCustomer(string name, string number) {

Customer cust = Container.NewTransientInstance<Customer>();

cust.Name = name;

cust.Number = number;

return Container.Persist<Customer>(ref cust);

}

}

Fixtures are registered by overriding the Fixtures property on the AcceptanceTestCase, for example:

protected override object[] Fixtures {

get {

return new object[] {new FixtureEntities(), new FixtureLinksUsingTypeName()};

}

}

The fixtures are installed by calling the RunFixtures() method, which will typically be done within the test initialization method, for example:

private static bool fixturesRun;

[TestInitialize()]

public void TestInitialize() {

InitializeNakedObjectsFramework(this);

if (!fixturesRun) {

RunFixtures();

fixturesRun = true;

}

StartTest();

}

(In the example code above, a static flag is used to ensure that the fixtures are run only once for the whole test class.)

### End to End Testing with Selenium [TBD. Is this still valid?]

When using Naked Objects, user-interface testing is not always necessary, or even valuable:

* If you are using only generic views you can rely on the fact that the generic Naked Objects user interface is extensively auto-tested, and therefore concentrate all your testing effort on the domain model itself, for example using XATs.
* If your customisation is limited to CSS then you can’t impact the functionality of the application; the worst you could do is render the application unusable by hiding elements.

However, if you have customised the user interface then it will be necessary to test any parts of the user interface impacted by those custom views or controllers - either relying on manual testing or through an automated UI test framework.

The generic Naked Objects user interface is tested, as part of its build process, using the Selenium framework. You can look up these tests within the Naked Objects source code (in the project NakedObjects.Spa.Selenium), and you might find it useful to copy the patterns used within those tests. In particular, the base class GeminiTest provides many helper methods for writing tests in a convenient form, as illustrated in the following example, which tests that clicking on a reference property opens a view of the object referenced in that property:

public virtual void ClickReferenceProperty() {

GeminiUrl("object?o1=\_\_\_1.Store--350&as1=open");

WaitForView(Pane.Single, PaneType.Object, "Twin Cycles");

var reference = GetReferenceProperty("Sales Person", "Lynn Tsoflias");

Click(reference);

WaitForView(Pane.Single, PaneType.Object, "Lynn Tsoflias");  
}

## How to use the cluster pattern

The Cluster Pattern is a specific high-level pattern for breaking up a large domain object model – following a very strict set of rules. Each cluster provides a distinct, re-usable, piece of business functionality. The term 'cluster' has been used only to distinguish this pattern from the more general idea of re-usable 'modules' or 'components' which, although they might have some features in common with the cluster pattern, typically do not enforce such strict rules.

The Cluster Pattern does not depend upon the Naked Objects Framework (NOF). However, the NOF makes it easier to implement the pattern, and makes its value more explicit.

#### Hard Rules

1. Each Cluster is defined by separate Api and Impl projects - the latter referencing the former. The Api defines only the programmatic interface to the Cluster (as may used by other Clusters) - it does not define the user view of the cluster. In general the Api exposes the minimum possible of the implementation.

2. A cluster depends on other clusters only where this clearly makes sense from a business perspective. For example, the Emails cluster depends upon the Documents cluster as the created emails may be stored as documents.

3. A Cluster may only ever reference the Api of another Cluster; it may never reference another Impl project. (It is recommended that this be enforced by build script rules on your build server).

4. A Cluster Api will typically consist of interfaces. An Api may also contain Enums, Constants, and (less commonly) static classes and methods. Members on interfaces should use only other interface types, Enums, or .NET value types. An Api may NOT contain any persistable classes, whether abstract or concrete. They may contain ViewModels or other non-persistent classes, though this is not encouraged as it can lead to confusion. It follows from the above that:

* Classes in a Cluster Impl may not inherit from classes in other clusters: this is a deliberate constraint.
* Any associations between objects in different clusters must be defined by interfaces, and must therefore follow a Polymorphic Association pattern.

5. The interfaces in a Cluster Api are explicitly labelled as one of three types:

* A **service** interface defines a service for which there is an implementation in the Impl. Other clusters may require an implementation of this service interface injected into them.
* A **result** interface is a restricted view of a domain type that is defined within the cluster’s Impl and may be created and/or retrieved by means of service interface methods. Good practice says that result interfaces should hide data behind higher-value behaviours where possible i.e. provide methods in preference to properties; any properties that *are* exposed should be read-only except in rare cases – in which case the rationale should be documented with comments.
* A **role** interface is intended to be implemented by objects in *other* clusters in order that those objects can take advantage of behaviour implemented in the cluster. Thus, those role interfaces may form input-parameters for methods on the service interfaces defined in the API. Additionally, the cluster Impl may define ContributedActions for that role interface.

1. Clusters should form a natural hierarchy of depenency: if the Api and Impl were treated as a single entity there should be no cicular dependencies between them. This is a subtle point, not the same thing as saying that there should be no circular dependencies between projects (which would not compile anyway). This is illustrated in the following diagrams. First, an example of a cluster hierarchy:



Notice that each cluster depends only on cluster below it. Expanding the view of three of the clusters, we can see that each one has an Api and Impl project; that the Impl depends on the Api, and that the Api *and/or* the Impl projects may depend on the Api’s in other clusters, but never on other impls:



Finally, here is a diagram of an earlier iteration of the same cluster project:



The diagramming tool (Visual Studio Ultimate in this case) has detected a circularity between clusters, shown in red. This is not strictly a circular reference between projects (as that would not compile). It occurs, in this example, because the Users.Impl project referenced Emails.Api, and Emails.Impl referenced Users.Api. in addition to the (intended) dependencies of Emails on Documents, and Documents on Users. The problem with this sort of circularity is that it means that Emails, Documents and Users cluster would always have to be used together and effectively form a ‘mega-cluster’. This error was easy to correct, however.

#### Optional Rules

The following represent good practices which are followed in this code base, but are not strictly definitional to the cluster pattern:

6. Each cluster defines its own DbContext and mappings.

7. Each cluster manages its own authorization via a standard pattern. The roles used by this authorizer are defined as constant strings on the cluster API.

8. Each cluster has its own test project, with emphasis on XATs that include full persistence, rather than on unit testing. (We follow Jim Coplien's advice that unit testing is best applied only to functions that have a clear, objective, and public, definition such as an algorithm.)

9. All projects in this library are set to All Warnings As Errors - so that they are warning free.

10. Clusters may be deployed as NuGet Packages, one each for the Api and Impl respectively, using SemVer rules for versioning.

### The Clusters project on GitHub

The open-source Clusters project on GitHub (see <https://github.com/NakedObjectsGroup/Clusters> ) provides an example cluster hierarchy that follows the rules specified above. Those clusters may be used and/or modified within a domain model project

**Important**: While the Clusters project does contain a significant number of tests, the test coverage is not comprehensive. If you use the code from the Clusters in your own project we recommend that you write additional tests as needed. Unlike the Naked Objects Framework, the Clusters project does not currently have a formal release process.

## How to build the framework from source

The source code for the Naked Objects Framework is hosted on GitHub here: <https://github.com/NakedObjectsGroup/NakedObjectsFramework>

To build the framework locally from source, open one of the top-level solutions such as Core.Sln. Ensure that you have the latest version of the NakedObjects.Ide NuGet package installed at solution level, as this installs a set of PowerShell scripts. To see these scripts, open the Package Manager Console window and type:

PM> get-help nakedobjects

Which will return:

Name Category Module Synopsis

---- -------- ------ --------

New-NakedObjectsCleanBuildTest Function Do a clean build of the Naked Objects Framework ...

New-NakedObjectsCleanBuildNoTest Function Do a clean build of the Naked Objects Framework ...

Get-NakedObjectsAllPackageVers... Function Display the versions of the NakedObjects package...

Update-NakedObjectsAllPackageV... Function Reads the new versions from nof-package-versions...

Update-NakedObjectsPackageVersion Function Update the version of a single NakedObjects pack...

Update-NakedObjectsPackageConf... Function Update all packages.config files and all .csproj...

You can obtain further help, including examples, for each of these functions by e.g.:

PM> get-help New-NakedObjectsCleanBuildNoTest

# Troubleshooting

## Logging

If you are hitting run-time errors, and the source of the error is not clear from any exception being thrown, then a recommending step is to take advantage of logging.

Naked Objects manages logging through the Common.Logging framework.

For an example of how to configure logging, look at the simple [Template project](#_Running_the_Template), which uses Log4Net as its logging implementation. The configuration is done in Web.config. The configuration in the Template project records logging messages into the file nakedobjects\_log.txt within the solution.

## Errors thrown when starting an application

This section contains hints and tips if your Naked Objects application throws an error during the start up process.

### No known services

This error indicates that you have not [registered](#_Registering_domain_services) any services.

### Unable to infer a key

This error should only occur if you are running Code First with Entity Framework. It indicates that the class FooBar does not have a key property. You need to ensure that the class has an integer property that can serve as the database key. If the key property follows the naming convention of [classname]Id or just Id then it will be picked up by the Code First mechanism automatically; if you do not wish to follow this convention, then you may instead mark up the property with the System.ComponentModel.Key attribute.

The best way to avoid this error occurring is to create new domain classes using the Domain Object item template, which automatically creates a key property.

Another possibility that can give rise to this error is if you have registered the class FooBar as a service, but it contains one or more properties. Services may not have properties. See [Service](#_Service).

### Class not public

This error was caused by the domain class FooBar not being public.

The best way to avoid this error occurring is to create new domain classes using the Domain Object item template, which automatically creates a public class.

## Errors thrown when running an application

This section contains guidance on error messages that might appear while you are using a Naked Objects application.

### A property is not virtual/overrideable

This error has arisen because the class FooBar contains:

* One or more properties that have not been made virtual. This is a requirement in order to allow the Entity Framework to create a proxy for the object. The best way to avoid this error occurring is to create all new properties using the propv code snippet.
* A collection that is not of type ICollection. See [Collection properties](#collection_property). The best way to avoid this error occurring is to create new collections using the coll code snippet.

### Invalid column name

This error is quite common when prototyping in Code First mode with Entity Framework - where a new property (in this case called Surname) has been added to a class since the database was created. The solution is either to delete the database and run again (which will re-create the database), or to manually add a new column to the appropriate database table.

### Invalid object name

As indicated by the fact that this is a SqlException, this error arises where a domain object class does not have a corresponding table in the database. The remedy is the same as for Invalid Column Name (above).

### Collection not initialised

This error has arisen because a collection was not initialised. See [Collection properties](#collection_property). The best way to avoid this error occurring is to create new collections using the coll code snippet.

### Database is generated, but certain (or all) tables are not being generated

This suggests that the framework is not identifying any domain classes. Two possible reasons for this are:

* Domain class namespaces beginning with NakedObjects. The Naked Objects framework assumes that any classes within this namespace are part of the framework, and ignored by the domain model reflector. This is to avoid accidentally treating framework classes as domain entities, and thus building them into the database schema. Choose another namespace for your domain models.
* Domain classes not reachable. Naked Objects builds its metamodel (which in turn is passed to Entity Framework to create the entity model and hence the database schema) by walking the graph from known start points - which means the set of [registered services](#_Registering_domain_services). So if there is no way to get to a particular class, either directly or indirectly, via any method on any registered service - then the class won't have been reflected on during the start-up phase.

## Unexpected behaviour in the user interface

This section contains suggestions on what to look for if your application is not behaving as you expect at the user interface.

### A public method is not appearing as an object action

Possible causes:

* Naked Objects does not recognise any overloaded methods as actions. (This is because the various overloaded versions would show up on the menu with the same label.)
* Naked Objects does not recognise templated methods as actions.
* Naked Objects only recognises as actions those methods with parameter types that are either [recognised value types](#_Recognised_Value_Types) or domain objects.

### Default, Choices, Validate or other complementary methods are showing up as menu actions

If you have written a Default, Choices, Validate or other 'complementary method' that is intended to work with a Property or Action, and it shows up as a menu action on the user interface then the method isn't being recognised as a complementary method by the Naked Objects framework. This is probably due to one or more of the following:

* The name of the helper method (following the prefix) does not exactly match the name of the Property or Action it is intended to assist. The difference might be in spelling or case.
* The parameters of the helper method differ in type from those of the Property or Action it is intended to assist.
* A complementary method in a sub-class will only be applied to a property or action defined in a super-class, if that complementary method is also defined on the super-class and overridden in the sub-class. To put that another way: complementary methods must be defined in the same class as the property or action they apply to, but may then be overridden in sub-classes.
* The name of the corresponding action happens to begin with one of the prefixes recognised by Naked Objects (for example: ClearComment). The best policy is to avoid using action names that begin with this prefixes - give the action a different name and then use the DisplayName attribute to change it back to what you want on the user interface.

## Debugging

This section deals with debugging your code.

### Life Cycle methods are not being called

If you suspect that an life-cycle method is not being called, then you should first verify this by inserting a break point on the method. If it is not being called as expected, then the most likely explanations are:

* The method is not public.
* The method is not void.
* The method name is mis-spelled
* The method name does not begin with an initial upper-case letter.

### The injected Container is null

The most likely explanation for this is that you have accidentally shadowed the property for the injected container in your domain class hierarchy. You can set Visual Studio to warn against this.

### An injected service is null

This may be for the same reason as the point above. If not, then the most likely explanation is that you haven't [registered the service](#_Configuring_the_list).