Naked Objects – Developer Manual

Contents

[Getting started 1](#_Toc395531463)

[Writing your first Naked Objects application 2](#_Toc395531464)

[Creating a domain model project 4](#_Toc395531465)

[Define your model project(s) 4](#_Toc395531466)

[Naked Objects IDE 4](#_Toc395531467)

[Define your DbContext(s) 5](#_Toc395531468)

[Overriding the default database schema generation 6](#_Toc395531469)

[Using data fixtures with Code First 7](#_Toc395531470)

[Running your domain model(s) as a Naked Objects MVC application 9](#_Toc395531471)

[End User Guide 10](#_Toc395531472)

[Service menus 11](#_Toc395531473)

[Action dialog 12](#_Toc395531474)

[The Find menu 13](#_Toc395531475)

[Programming Reference 15](#_Toc395531476)

[Programming concepts 16](#_Toc395531477)

[Domain object 16](#_Toc395531478)

[Property 16](#_Toc395531479)

[Action 17](#_Toc395531480)

[Recognised method 18](#_Toc395531481)

[Recognised attribute 18](#_Toc395531482)

[View Model 18](#_Toc395531483)

[Run class 21](#_Toc395531484)

[Service 22](#_Toc395531485)

[Repository 23](#_Toc395531486)

[External or System service 26](#_Toc395531487)

[Contributed action 27](#_Toc395531488)

[Dependency Injection 29](#_Toc395531489)

[The Domain Object Container 30](#_Toc395531490)

[Entity Persistor 31](#_Toc395531491)

[Data fixture 32](#_Toc395531492)

[Optional vs. Required 34](#_Toc395531493)

[The Naked Objects programming model 35](#_Toc395531494)

[Recognised Value Types 35](#_Toc395531495)

[Recognised Collection types 35](#_Toc395531496)

[Recognised .NET attributes 36](#_Toc395531497)

[Recognised Methods 39](#_Toc395531498)

[NakedObjects.Attributes 42](#_Toc395531499)

[NakedObjects.Types 56](#_Toc395531500)

[NakedObjects.Helpers 57](#_Toc395531501)

[Adding behaviour to your domain objects - a how-to guide 59](#_Toc395531502)

[Using the Naked Objects IDE 60](#_Toc395531503)

[Item Templates 60](#_Toc395531504)

[Code Snippets 60](#_Toc395531505)

[The object life-cycle 63](#_Toc395531506)

[How to create an object 63](#_Toc395531507)

[How to persist an object 64](#_Toc395531508)

[How to update an object 64](#_Toc395531509)

[How to delete an object 64](#_Toc395531510)

[How to retrieve existing instances 65](#_Toc395531511)

[How to insert behaviour into the object life cycle 65](#_Toc395531512)

[How to specify that an object should never be persisted 65](#_Toc395531513)

[How to specify that an object should not be modified by the user 65](#_Toc395531514)

[How to specify that a class of objects has a limited number of instances 65](#_Toc395531515)

[How to implement concurrency checking 65](#_Toc395531516)

[Object presentation 66](#_Toc395531517)

[How to specify a title for an object 66](#_Toc395531518)

[How to specify the icon for an object 67](#_Toc395531519)

[How to specify a name and/or description for an object 67](#_Toc395531520)

[How to specify that an object should be always hidden from the user 68](#_Toc395531521)

[Properties 69](#_Toc395531522)

[How to add a property to a domain object 69](#_Toc395531523)

[How to prevent the user from modifying a property 69](#_Toc395531524)

[How to make a property optional (when saving an object) 70](#_Toc395531525)

[How to specify the size of String properties 70](#_Toc395531526)

[How to validate user input to a property 70](#_Toc395531527)

[How to validate user input to more than one property 71](#_Toc395531528)

[How to specify a default value for a property 72](#_Toc395531529)

[How to specify a set of choices for a property 72](#_Toc395531530)

[How to specify auto-complete for a property 74](#_Toc395531531)

[How to set up the initial value of a property programmatically 74](#_Toc395531532)

[How to trigger other behaviour when a property is changed 74](#_Toc395531533)

[How to control the order in which properties are displayed 75](#_Toc395531534)

[How to specify a name and/or description for a property 75](#_Toc395531535)

[How to hide a property from the user 75](#_Toc395531536)

[How to make a property non-persisted 76](#_Toc395531537)

[How to handle File Attachments 77](#_Toc395531538)

[How to display an image 79](#_Toc395531539)

[How to handle enum properties 79](#_Toc395531540)

[Collection properties 81](#_Toc395531541)

[How to add a collection property to a domain object 81](#_Toc395531542)

[Adding-to or removing objects from a collection 81](#_Toc395531543)

[How to create a derived collection 82](#_Toc395531544)

[How to control the order in which table rows are displayed 83](#_Toc395531545)

[How to specify which columns are displayed in a table view 83](#_Toc395531546)

[Actions 84](#_Toc395531547)

[How to add an action to an object 84](#_Toc395531548)

[How to define a contributed action 84](#_Toc395531549)

[How to prevent a service action from being a contributed to objects 84](#_Toc395531550)

[Hor to specify parameter names and/or descriptions 84](#_Toc395531551)

[How to make a parameter optional 84](#_Toc395531552)

[How to specify a default value for a parameter 84](#_Toc395531553)

[How to specify a set of choices for a parameter 85](#_Toc395531554)

[How to allow selection of multiple choices 87](#_Toc395531555)

[How to specify auto-complete for a parameter 87](#_Toc395531556)

[How to specify the length or format for text-input parameters 88](#_Toc395531557)

[How to obscure input text (e.g. for a Password) 88](#_Toc395531558)

[How to validate parameter values 88](#_Toc395531559)

[How to specify conditions for invoking an action 89](#_Toc395531560)

[How to control the order in which actions appear on the menu 90](#_Toc395531561)

[How to hide actions 91](#_Toc395531562)

[How to pass a message back to the user 91](#_Toc395531563)

[How to throw an application exception 92](#_Toc395531564)

[How to work with transactions 92](#_Toc395531565)

[Advanced Entity Framework techniques 93](#_Toc395531566)

[How to handle concurrency checking 93](#_Toc395531567)

[How to specify 'eager loading' of an object's reference properties 93](#_Toc395531568)

[How to avoid losing your code modifications 94](#_Toc395531569)

[How to implement complex types 95](#_Toc395531570)

[How to work with multiple databases 96](#_Toc395531571)

[How to handle associations that are defined by an interface rather than a class 96](#_Toc395531572)

[How to work with multiple database contexts 97](#_Toc395531573)

[Customising the MVC User Interface 100](#_Toc395531574)

[Customising the CSS 100](#_Toc395531575)

[Custom Views 101](#_Toc395531576)

[Custom controllers 106](#_Toc395531577)

[MVC - Additional How-Tos 113](#_Toc395531578)

[How to inject Services into Controllers and Views 113](#_Toc395531579)

[How to encrypt hidden fields in a transient object 113](#_Toc395531580)

[How to disable server-side validation (Ajax) 114](#_Toc395531581)

[Other 115](#_Toc395531582)

[Getting hold of the current user programmatically 115](#_Toc395531583)

[Creating an XML Snapshot of an object 116](#_Toc395531584)

[Creating and using a Restful Objects API 117](#_Toc395531585)

[Adding a Restful Objects API to a Naked Objects MVC application 118](#_Toc395531586)

[Creating a standalone Restful Objects API 119](#_Toc395531587)

[Conformance to the Restful Objects specification 120](#_Toc395531588)

[Custom extensions 120](#_Toc395531589)

[Restful Objects Server - additional how-to's 121](#_Toc395531590)

[How to determine whether an action will require a GET, PUT or POST method 121](#_Toc395531591)

[How to work with proto-persistent objects 121](#_Toc395531592)

[How to specify the scheme for domain model type information 122](#_Toc395531593)

[How to enforce concurrency-checking 122](#_Toc395531594)

[How to make an application Read Only 122](#_Toc395531595)

[How to implement automatic redirection for specific objects 123](#_Toc395531596)

[How to change cache settings 123](#_Toc395531597)

[How to handle Complex Types 124](#_Toc395531598)

[How to change the format of the Object Identifier (Oid) in resource URLs 125](#_Toc395531599)

[How to limit the scope of the domain model that is visible through the Restful API 126](#_Toc395531600)

[How to switch off strict Accept header enforcement 127](#_Toc395531601)

[How to limit the size of returned collections 127](#_Toc395531602)

[How to enable cross-origin resource sharing (CORS) 128](#_Toc395531603)

[How to work with addressable View Models 129](#_Toc395531604)

[Authorization in Restful Objects 129](#_Toc395531605)

[Running without a user interface 131](#_Toc395531606)

[Running Naked Objects as a .exe 132](#_Toc395531607)

[Running multiple threads asynchronously 133](#_Toc395531608)

[Authorization 134](#_Toc395531609)

[Attribute-based Authorization 135](#_Toc395531610)

[Custom Authorization 137](#_Toc395531611)

[Controlling visibility of columns in a table view 138](#_Toc395531612)

[WIF Claims-based authorization 139](#_Toc395531613)

[Auditing 141](#_Toc395531614)

[Testing 142](#_Toc395531615)

[Executable Application Tests (XATs) 143](#_Toc395531616)

[Creating an XAT test class 143](#_Toc395531617)

[Writing tests 144](#_Toc395531618)

[Simulating users and roles 146](#_Toc395531619)

[Running tests against a database 146](#_Toc395531620)

[End to End Testing with Selenium 148](#_Toc395531621)

[Adding a Selenium test project 148](#_Toc395531622)

[Writing Selenium tests 148](#_Toc395531623)

[Internationalisation / Localisation 151](#_Toc395531624)

[Adding the Resources projects to your solution 152](#_Toc395531625)

[Specifying that you wish to localise the application 153](#_Toc395531626)

[Populating the Model resource file 154](#_Toc395531627)

[Translating the resource files 155](#_Toc395531628)

[Testing your localised application 157](#_Toc395531629)

[Troubleshooting 158](#_Toc395531630)

[Logging 159](#_Toc395531631)

[Problems running the AdventureWorks example application 160](#_Toc395531632)

[Error locating server 160](#_Toc395531633)

[The application runs very slowly 160](#_Toc395531634)

[Problems working CodeFirst 161](#_Toc395531635)

[Database is generated, but certain (or all) tables are not being generated 161](#_Toc395531636)

[Errors thrown when starting an application 162](#_Toc395531637)

[No known services 162](#_Toc395531638)

[Unable to infer a key 162](#_Toc395531639)

[Class not public 162](#_Toc395531640)

[Errors thrown when running an application 163](#_Toc395531641)

[A property is not virtual/overrideable 163](#_Toc395531642)

[Invalid column name 163](#_Toc395531643)

[Invalid object name 163](#_Toc395531644)

[Collection not initialised 163](#_Toc395531645)

[Could not find Entity Framework context for type 163](#_Toc395531646)

[Unexpected behaviour in the user interface 164](#_Toc395531647)

[A public method is not appearing as an object action 164](#_Toc395531648)

[Default, Choices, Validate or other complementary methods are showing up as menu actions 164](#_Toc395531649)

[Debugging 165](#_Toc395531650)

[Life Cycle methods are not being called 165](#_Toc395531651)

[The injected Container is null 165](#_Toc395531652)

[An injected service is null 165](#_Toc395531653)

[The application runs OK in execute mode, but throws an exception in debug mode 165](#_Toc395531654)

## 

# Getting started

If you’re new to Naked Objects start by trying this ultra-simple example: [Writing your first Naked Objects application](#_Writing_your_first). Then you can progress in a slightly more formal manner.

A Naked Objects application will typically consist of multiple projects.

First, there will typically be one or more ‘Model’ projects, containing your domain classes. See:

[Creating a domain model project](#_Creating_a_domain_1)

Next you will need one or more ‘Run’ projects. If you want a ready-made user interface then you should add an MVC project. See:

[Running your domain model(s) as a Naked Objects MVC application](#_Running_your_domain)

Later you might also decide to add a Restful API, as an alternative way to ‘run’ your domain model:

[Creating and using a Restful Objects API](#_Writing_a_web-application)

Or the ability to run as a standalone executable, for example for Batch operations:

[Running without a user interface](#_Running_without_a)

You will also want to add one or more ‘Test’ projects. These may be conventional Unit Tests, but we strongly recommend that you also consider using the powerful Naked Objects XAT framework, to write functional or integration tests that are UI-independent:

[Executable Application Tests (XATs)](#_Executable_Application_Tests)

If you are sticking to the generic Naked Objects MVC User Interface then there is typically no need to then write UI tests (which are notoriously brittle to change). But if you customise the user interface (other than just by modifying the CSS) then you should write UI tests. We’ve written some helper classes to work with the respected Selenium framework, see:

[End to End Testing with Selenium](#_End_to_End)

## Writing your first Naked Objects application

Follow these steps to write your first ultra-simple Naked Objects application, featuring one simple domain class only:

1. Using Visual Studio 2013 (Express version is fine) create a new *ASP.NET Web Application* project called, say, *MyWebApp*, using the *MVC* template.
2. Install the NuGet Package *NakedObjects.Mvc-FileTemplates*, selecting *Yes To All* when asked if you wish to overwrite existing files.
3. In the Models folder add a new class Customer as follows. Note that:

* All properties in a Naked Objects application must be virtual.
* [Hidden] specifies that this property is not for display on the user interface.
* [Title] specifes that the value of the property should be displayed in the Tab.

using NakedObjects;

namespace MyWebApp

{

public class Customer

{

[Hidden]

public virtual int Id { get; set; }

[Title]

public virtual string Name { get; set; }

}

}

1. Create a DbContext object as follows. (This is standard Entity Framework Code First coding.)

using System.Data.Entity;

namespace MyWebApp

{

public class MyDbContext : DbContext

{

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

}

}

1. In the *App\_Start* folder find the RunWeb class and edit two members as follows. First, the MenuServices property which defines the services to be shown on the main menu. (NakedObjects.Services.SimpleRepository is a ready-made class for early-stage prototyping only.)

protected override IServicesInstaller MenuServices {

get {

return new ServicesInstaller(

new SimpleRepository<Customer>());

}

}

Second, the Persistor property, in which we need to specify the DbContext(s) that it needs to inspect:

protected override IObjectPersistorInstaller Persistor

{

get

{

var installer = new EntityPersistorInstaller();

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

return installer;

}

}

1. Run the project. Using the actions on the *Customers* menu, try creating and retrieving Customer objects.

## Creating a domain model project

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 (in Naked Objects MVC) or Restful API (Restful Objects for .NET) the Naked Objects framework treats all properties as *mandatory*, unless marked up with the Optionally attribute - as we believe that this is the safer default behaviour. Note that it is possible to configure Naked Objects to work the other way if you wish (see [Optional vs. Required](#_Optional_vs._Required)).

### Define your model project(s)

Your starting point will be to develop one or more Model projects, as follows:

1. Create a new Class Library project.
2. Invoke Manage NuGet Packages, find and install the NakedObjects.ProgrammingModel package (see also note below about Naked Objects IDE)
3. Add POCO domain classes, following the simple programming conventions that are recognised by the Naked Objects framework. See [Domain object](#_Domain_object).
4. Create some services to act as Repositories/Factories.

### Naked Objects IDE

When you installed the NakedObjects.ProgrammingModel via the NuGet Package Manager, it automatically added the NakedObjects.Ide package, which consists of a set of Item Templates and Code Snippets that can be very useful when construction your domain model. The item templates are used when invoking Add > New Item: you will find them listed under Naked Objects for the C# programming language. To see the list of code snippets: Tools > Code Snippets Manager > [Language] > My Code Snippets > Naked Objects. Most can be invoked using their shortcut, such as 'propcho' to add a Choices method for a property. Note, however, that if you have other tools installed, such as Resharper, then these can sometimes invalidate the shortcuts. The IDE has been placed in its own package so that you have the option when un-installing the NakedObjects.ProgrammingModel from a specific project, whether or not you want to uninstall the IDE, which may be used by multiple projects.

### Define your DbContext(s)

Next you will need to set up a 'Context' class for your model project, which is defined in exactly the same way as for conventional Code First development. You can use the Naked Objects > DbContext item template to help create this class - as shown in the example below:

namespace DataAccess

{

public class MyContext : DbContext

{

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

public MyContext() { }

public DbSet<Foo> Foos { get; set; }

public DbSet<Bar> Bars { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

//Initialisation

//Use the Naked Objects > DbInitialiser template to add a custom initialiser, then reference thus:

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

//Mappings

//Use the Naked Objects > Mapping template to add mapping classes & reference them thus:

//modelBuilder.Configurations.Add(new Employee\_Mapping());

}

}

}

The context inherits from the class System.Data.Entity.DbContext it defines methods to return one or more DbSets.

It is not necessary to define a DbSet for each of your domain classes - just for the 'root' classes in your model hierarchy.

Within the OnModelCreating method you may also:

* Add database mappings using the Code First 'Fluent API'. Use the Naked Objects > DbMapping item template to create a mapping class quickly.
* Add a database initialiser to determine when/whether to drop and re-create the schema. Use the Naked Objects > DbInitialiser item template to create an initialiser class quickly. You can use this initializer to seed the database, but you should consider - as an alternative - using Naked Objects' own Data Fixtures pattern as this has several advantages,

### 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");

}

}

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, one option for creating data fixtures (to pre-populate the database with) is 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" });

//...

}

}

}

This custom database initializer may be set within the Persistor property of the Run class, as described earlier in this section, or within the OnModelCreating() method on your DbContext.

Another option is to use Naked Objects' own Data Fixtures pattern, which has the advantage that you can delegate to methods on services and objects just as if they were created by the user.

## Running your domain model(s) as a Naked Objects MVC application

Follow these steps to run your domain model(s) with the Naked Objects MVC user interface.

1. Create a new C# ASP.NET Web Application project, specifying MVC as the template,and set this as the Startup Project for your solution.
2. Invoke Manage Nuget Packages, find and install the NakedObjects.Mvc-FileTemplates package into this project
3. Add project reference(s) to your Model project(s).
4. Copy the connection string(s) from the App.config file(s) in your Model project(s) into the Web.config in your Run project.

There are two web.config files in a standard MVC project - one within the Views folder (the purpose of which is to prevent direct access to the views without going through a controller) and the other at the project root level. The connection string needs to go into the latter.

1. Open the RunWeb class (within the App\_Start folder). Within this Run class, register your services, fixtures (if required), and make any other changes you may require to the default run configuration.
2. Run the project as a local app. This will launch a web-browser pointing at the start page of your application, running within LocalHost.
3. Even if you selected the Internet project template, by default, the application is not initially set up to require a log-on. When you are ready to add authorization to your prototype, simply un-comment the commented-out the [Authorize] attribute on the GenericController and SystemController classes.
4. When you are ready, you can deploy the application to a server running IIS, in exactly the same way as any other ASP.NET MVC application. The simplest way is to right-click on the project and invoke the Publish action.

# End User Guide

This section describes the generic 'out-of-the box' UI that you will get when using the default .css styling.

## Service menus

Each page features a bar across the top representing the various services available to the user, each of which will have a drop-down menu of service actions. Some of these actions will execute immediately, typically returning one or more domain objects. Other actions will return ....

## Action dialog

An action dialog will have one or more parameters to be completed, and an OK button. (To cancel the dialog, without executing the action, simply use the back button on the browser). Mandatory parameters are indicated by a red asterisk, and the OK button will automatically be disabled until these are completed. On hitting OK, if the value of any parameter fails validation rules, explanatory messages will be displayed adjacent to the parameter and/or at the top of the dialog.

Parameters may be divided into two types: value and reference. Value parameters - which include strings, numbers and dates - may be typed in to the field, though there may also be helper mechanisms such as a date picker displayed.

Reference parameters require a reference to a domain object. You cannot type into those fields. For some references parameters the programmer will have specified a set of choices for the parameter, which will be rendered as a drop-down list. In other cases, the user may supply any object of the correct type. In these cases a Find menu alongside the entry field will provide the user with actions for Finding (or even creating) a domain object of the appropriate type (these actions will typically be a sub-set of the actions available on the service menus). Several of these actions may cause a separate dialog to be opened-up within the parameter field. The top-most option on the Find menu is called Recently Viewed. This offers the user a list of objects of the appropriate type that have been recently displayed on the screen.

## The Find menu

A Find menu, is rendered wherever there is a requirement for the user to specify a domain object - either as a parameter within an action dialog, or as a property within an object being edited - wherever the user is not presented with an explicit set of choices (specified either via a Choices method, or by a Bounded class). An example is shown below:



In the above example the Find menu has been rendered on the Customer parameter within an action dialog. The top-most option - Recently Viewed - appears on all instances of the Find menu. It will return a small set of objects that are of the required type and that have recently been displayed to the user. This set is continuously updated.

If the type of the parameter of property is defined by an Interface, then the Recently Viewed option will display recently-viewed objects of any type that implements that interface.

The other options on the Find menu will be service actions that return an object, or collection of objects, of the right type. These will be grouped by the service name. In the above example, the actions that return one or more Customer objects are all to be found on the repository (service) called Customers and so are rendered within a sub-menu of that name.

Invoking one of the Find menu actions will result directly in one of three things, as illustrated in the following screen shots. The first case is that the action will directly return one or more objects with a Select button next to each. There will also be a Details link to allow inspection of the object - use the browser's Back button to return to the previous page and select the object (or invoke another action from the Find menu). Hitting Select will place a reference to the selected object within the required field.



The second case is where the Find menu action requires parameters, In this case a dialog will be rendered in situ.



The Find menu will not include any action that would ordinarily include a Find menu on one of its parameters. This is to avoid ending up with a Find dialog inside a Find dialog - which would be potentially very confusing to a user, and very difficult to implement as an HTML form.

Selecting OK will render the returned object(s) as shown previously.

The third case is where the Find menu action is to create a new object. Here the user will be presented with an unsaved object to be completed and saved, and then to be selected as before.

If the unsaved object requires one or more reference object properties to be specified, and these do not have an explicit set of choices, then it will not be possible to proceed - for the same reason given above that we cannot end up with Find menus inside Find menus. In this situation (which should be a rare occurrence) the action method should be marked up with the ExcludeFromFindMenu attribute.

# Programming Reference

## 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, or nouns, 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.

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 as a shortcut.

#### Value Property

A 'value property' has a string, number, date, or other recognised value type. 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. (Certainly 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 (as an icon and title), to which the user may navigate. In edit mode, the user may specify the object to be associated by a number of mechanisms such as 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. However, it is recommended that you specifically use ICollection<T>. Collections must be initialised. 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. Note also that static methods are ignored by Naked Objects.

Tip: Use the Action snippet (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 Hidden attribute
* 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: Section 2.2.7, “Contributed action”.

When a method returns a reference the viewer will attempt to display that object. If the return value is Nothing / Null then nothing is displayed.

#### Actions that return a collection

An action (on a service or a domain entity object) may return any of the 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. 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 not be shown.

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.

### 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.

The use of recognised methods is described in Section 2.3, “Adding behaviour to your domain objects - a how-to guide” or you may view the complete list of: Section 2.4., “Recognised Methods”.

### Recognised attribute

Naked Objects recognises a set of attributes that may optionally be applied to domain objects, properties, or actions, to alter presentation or behaviour. Their use is described in Section 2.3, “Adding behaviour to your domain objects - a how-to guide” or you may view the complete list of: Section 2.4.5, “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 and IViewModelEdit

In Naked Objects, a View Model is coded the same way as a domain object but must implement the interface NakedObjects.IViewModel, (or one of its two sub-types IViewModelEdit and IViewModelSwitchable – more on these shortly) which requires these two methods to be 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.
* PopulateUsing 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 PopulateUsing(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 persistent Customer, if you don't want it. The view model could simply hold the Customer's Id.

In Naked Objects MVC view models are rendered using an ObjectView.aspx (or .cshtml) views, like any other object - either the generic one in the Views > Shared folder, or a custom one located in a folder named by the type of the view model.

**IViewModelEdit and IViewModelSwitchable**

IViewModelEdit is a specialised sub-type of IViewModel, with the only difference being that properties will be rendered as input fields (unless they are deliberately disabled) - and that the values will be taken on by the object when any action is invoked. In Naked Objects MVC, an IViewModelEdit will be rendered using the ViewModel.aspx (or .cshtml) view, not (as might be imagined) ObjectEdit. The reason for this is that the ObjectEdit view automatically renders a Save button, which is not applicable here. 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 (for example it might use the entered values to create one or more persistent objects).

IViewModelSwitchable allows a view model to be rendered either in view mode or in edit mode, depending on the value returned by the IsEditView() method that must then be provided. In Naked Objects MVC, an IViewModelSwitchable will be rendered using the ViewModel.aspx (or .cshtml) view when it is in edit mode, or using an ObjectView.aspx (or .cshtml) view when it is not in edit mode.

**Actions on Editable View Models - Limitations**

Editable view models were designed to support actions, with the principal intent of implementing multi-step processes or wizards - so you could add actions for .e.g Next, Previous, Start, Finish. This is why - with the default .css - actions on editable view models are rendered as buttons rather than a drop-down menu. When you click on such an action, any values you have entered into the editable fields are submitted to the server; the view model re-created on the server; and its properties then pdated with the values entered by the user before the action method is invoked (whether or not the action needs them).

The limitation is that this does not apply to actions that take parameters - and which thus return a modal dialog. In this case, when you hit OK on the dialog, the form sent back to the server is the dialog form, containing values you entered into the dialog NOT the form representing the fields on the underlying editable view model. As a secondary effect of this, if the action-with-parameters is void then when the underlying ViewModel is re-displayed, any values entered into that form will have been lost.

**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;

}

}

}

**Concurrency Checking and View Models**

For all un-editable 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.

### Run class

Every Naked Objects application requires a Run class, which serves as the start-up object for the application.

Note: For Naked Objects MVC, the Run class is located in the App\_Start folder of the run project.

The Run class is the place where you register the services that the application requires. This is done within one of three overridable properties on the Run class, according to how you want you want to do with the services:

* MenuServices. Services registered via this property will appear on the service menu bar in the user interface. These services will also be injected into objects that specify them. Any action methods on these services that take domain types as parameters will also be contributed as object actions to those domain types, unless marked up with the NotContributedAction attribute.
* SystemServices. Services registered via this property are for use internally by the system via dependency injection. The services will not appear on the user interface menu bar, and none of their methods will appear as contributed actions.
* ContributedActions. Services registered via this property can provide actions that will be contributed to appropriate domain objects, but do not appear on the service menu bar. They will also be injected where needed.

Warning: A service should not be registered in more than one of the three properties.

Each of these three properties is of type IServicesInstaller, and the simplest way to implement this is to return a new ServicesInstaller, passing in a new instance of each various services required as a parameter array, as illustrated below:

protected override IServicesInstaller MenuServices {

get {

return new ServicesInstaller(new SimpleRepository<Customer>(),  
 new MyOwnOrderRepository());

}

}

#### Configuration options in the Run class

In addition to the responsibility for launching the Naked Objects framework and for specifying the services, the Run class may also be used to configure the application. (The various Run...Base classes merely provide predefined configurations). Each of these options is specified via an Overridable property on the Run class. The most commonly-used options are listed below, with links to the sections explaining their use:

* [Entity Persistor](#_Entity_Persistor)
* [Data fixture](#_Data_fixture)
* [Optional vs. Required](#_Optional_vs._Required)
* [Custom Authorization](#custom_authorization)
* [Auditing](#_Auditing)

### 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 'repositories'. (Some designers prefer to draw a distinction between a 'factory' service, which has responsibility for creating domain objects, and a 'repository' service for retrieving them. If you prefer to maintain this distinction, then they may be implemented in two separate services. However, there is merit in keeping the ideas together, and we tend to refer to the combination as a 'repository'. For more information see Section 2.2.5, “Repository”.
* To provide a bridge to external functionality. For more information see Section 2.2.6, “External or System service”.
* 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. For more information see Section 2.2.7, “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 in the Services property of the Run class. Registering the service serves two purposes:

* It makes the service's actions available to the user
* It instructs the Naked Objects framework to inject that service into any domain object that needs access to it. See [Dependency Injection](#_Dependency_Injection).

A service may provide multiple methods any of which may appear as user actions - following the same rules as for actions on a domain object.

Warning: adding properties to a service - for example to cache retrieved objects in memory - can result in unreliable behaviour and hard-to-diagnose errors.

### Repository

In Naked Objects, a 'repository' is a service that provides methods for:

* Finding one or more domain objects where you don't have an associated object to navigate from, such as finding a Customer object by their name or customer number.
* Creating a new instance of a domain object class, where you don't have an existing object to create it from. Thus, although you might decide that it makes sense to create a new Order object by means of an action on Customer, you will probably want to be able to create a new Customer object without necessarily having any Order.

Strictly speaking the first role is known as a 'factory' and the second as a 'repository'. If you prefer to maintain this distinction, then they may be implemented in two separate services. However, there is merit in keeping the ideas together, and we tend to refer to the combination as a 'repository'.

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.

During the earliest stages of prototyping you can make use of the ready-made SimpleRepository class. As your design evolves, however, you will want to write your own repositories. It can be convenient (though it is certainly not necessary) to inherit from AbstractFactoryAndRepository.

#### SimpleRepository

Naked Objects provides a ready-made class SimpleRepository that is very useful during the earlier stages of prototyping an application - as the prototype progresses, these will typically be replaced with custom-written repositories.

Simple repositories are created within the Services property of the Run class: the code below shows a SimpleRepository being created and registered for the Customer class and the Order class:

protected override IServicesInstaller MenuServices {

get

{

return new ServicesInstaller(

new SimpleRepository<Customer>(),

new SimpleRepository<SalesOrderHeader>());

}

}

In the above example, the two simple repositories will appear on the menu bar of the application as Customers and SalesOrderHeaders respectively. If you wish to specify another name, such as Orders then include this name as a string parameter in the constructor, thus:

new SimpleRepository<SalesOrderHeader>("Orders")

The simple repository provides the user with a standard menu of actions as follows:

* New Instance. This action returns an unsaved object for the user to complete and save (or discard).
* All Instances. This action is intended purely for prototyping purposes, and returns all the current instances of the specified type, up to a maximum of 20.
* Get Random. Returns a randomly-selected instance of the specified type (very useful when giving early demonstrations and you've forgotten all the example data!)
* Find By Title. Searches for the given string within the Title of the instances - case insensitive. Note that this method only works for objects where one of the properties has been marked up with a Title attribute. It does not work on objects where the title is specified by a Title method.

#### Inheriting from AbstractFactoryAndRepository

The simplest way to create a Repository is to inherit from NakedObjects.Services.AbstractFactoryAndRepository. You can use the Factory and Repository template to create a sub-class of AbstractFactoryAndRepository with some helpful code formatting and comments using

AbstractFactoryAndRepository provides a number of methods that you can call from within your own methods, of which the two most commonly used are:

* NewTransientInstance 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.
* 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.

Since the result is queryable, you may query it using LINQ to obtain more specific matches.

The helper method SingleObjectWarnIfNoMatch returns the first item from the IQueryable and converts it to an object of the corresponding type. If the IQueryable contained no items, a warning message will be given to the user as illustrated in this example:

public Employee FindEmployeeByName(string name) {

var query = from employee in Instances<Employee>()

where employee.Name == name

select employee;

return SingleObjectWarnIfNoMatch(query);

}

Tip: Use the Find Query snippet (shortcut: find) to create a method that finds a single matching instance. Use the List Query snippet (shortcut: list) to create a method that returns a list of matching instances.

AbstractFactoryAndRepository also provides methods to provide capabilities equivalent to the actions provided by SimpleRepository:

* Random returns a random instance of a specified type of object.
* FindByTitle - finds objects of a specified type where the title includes the string provided (case insensitive). Note that this works only for objects where the title is specified using a Title attribute; it will not work where the title is specified by a Title() or ToString() method. There is a non-templated variant on this method, FindByTitleAndType where the type is specified as parameter.

The use of these methods is illustrated in the following example code (note the use of the DisplayName to give the repository a more friendly name when presented to the user):

[DisplayName("Customers")]

public class CustomerRepository : AbstractFactoryAndRepository

{

public CustomerRepository RandomCustomer()

{

return Random<Customer>();

}

public IQueryable<Customer> FindCustomerByTitle(string match)

{

return FindByTitle<Customer>(match);

}

}

#### Writing a Repository from scratch

If you are unable or unwilling to inherit from AbstractFactoryAndRepository then you can write your own repository from scratch. You simply need to provide a write-only property into which the framework can inject a Domain Object Container, 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.

### External or System service

The second role that services perform within a Naked Objects application is as a bridge to external functionality, we . 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 preexisting 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.
* (Less commonly) Bridging between isolated modelling domains. The Naked Objects philosophy is to aim, where possible, for a single coherent enterprise object model running within the same application space. Where this is not possible then services may be used to communicate between the domains without requiring common object definitions and/or identities.

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);

}

}

Tip: We recommend that you register any external services in the SystemServices property of the Run class. That way, they will not be offered to the user on the services menu.

### 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 - typically within a sub-menu that takes the name of the service where the action is defined.

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.

Tip: If you have one or more actions that you wish to appear as a contributed action, but not on a service menu, then place those actions in one or more separate service classes (we recommend including the phrase ContributedActions in the class name - though this is purely for readability, it has no functional effect) and register those classes in the ContributedActions property in the Run class.

Tip: If you have an action on a service that you wish to be available to the user on a service menu, but not as a contributed action on a domain object, then mark up that action with the NotContributedAction attribute.

In a more complex application, it might well be that a domain object might have several contribute sub-menus, each containing several methods. Designing an application this way allows us to keep the model well partitioned.

#### Actions contributed to individual objects

If an action defined on any service takes a domain object as one of its parameters, then that action will automatically be contributed to any domain object of that type - in a sub-menu that takes the name of the service.

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

[DisplayName("Tasks")]

public class TaskContributedActions : AbstractService

{

public Task CreateNewTask(ITaskContext context)

{

Task task = 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 parameters, but with the domain object representing one of those 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. 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 a 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(IQueryable<Transaction> transactions)

{

foreach (Transaction t in transactions.ToList())

{

t.MarkAsReconciled();

}

}

The following limitations apply to collection-contributed actions:

* Collection-contributed actions may not themselves return a collection. This is because the stateless architecture of Naked Objects MVC would not allow the resulting collection to be re-created automatically. 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 currently possible to make a selection that spans multiple pages of a query result. However, it is possible to expand the page size.

### Dependency Injection

'Dependency Injection' is a pattern whereby objects are automatically provided with references to any services that they need to make use of - in contrast to the 'Service Locator' pattern, where it is the responsibility of the object to locate the services it needs. Naked Objects has in-built dependency injection. It should be used wherever a domain object needs to access a Repository, or any other kind of service, programmatically (from within a method on the domain object). It may also be used where a Repository (or any other kind of service) needs to access methods from another service.

To use dependency injection, simply provide a property that specifies the type of service required. The property should have a protected get. 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 myProductRepository.ListProductsForColour(this.Colour);

}

}

Tip: Use the injs snippet to create this code.

Note: You can also use dependency injection to gain access to the Domain Object Container directly. See [IDomainObjectContainer methods](#_IDomainObjectContainer_methods).

### 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. For the list of methods available on this container, see Section 2.2.9.1, “IDomainObjectContainer methods”

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.:

Customer 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.

* NewTransientInstance 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(object). Note that 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.
* Principal returns a System.Security.Principal.IPrincipal that represents the logged on user. From this IPrincipal it is possible to object the name and/or roles of the user.
* 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.
* 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.

### Entity Persistor

The 'Entity Persistor' is the means by which Naked Objects stores object state on a relational database - making use of Microsoft's Entity Framework. Use of this Persistor is specified on the Run class as follows:

protected override IObjectPersistorInstaller Persistor {

get {

return new EntityPersistorInstaller();

}

}

The EntityPersistorInstaller can be configured to operate either in 'code-first' mode, or with a .edmx file.

Note: Naked Objects is not tied to Entity Framework - it is possible to create alternative persistence mechanisms. In earlier versions of Naked Objects we provided two such mechanisms - an 'in-memory persistor' and an 'xml persistor', both of them intended for prototyping. However, with the advances in Entity Framework's capabilities (especially 'code first' mode), the latter is now so easy to use - even for prototyping - that we no longer encourage the use of those older prototyping persistors. The 'xml persistor' has been moved onto NakedObjects.Contrib. The in-memory persistor still exists, and may be installed using the NakedObjects.Persistor.Objectstore.Inmemory.InMemoryObjectPersistorInstaller.

### Data fixture

Fixtures are used to set up objects within the code base, principally for use within prototyping and/or testing.

Naked Objects provides a dedicated pattern for creating and installing fixtures, to be run, automatically, each time you run the application. In this pattern, a fixture may be any class that has an Install method. The following code illustrates a simple fixture class:

public class CustomerFixture {

public IDomainObjectContainer Container {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 need to be registered in order for the framework to pick them up at run time. The most convenient way to do this within the Fixture property of your Run class as in the following example

protected override NakedObjects.Core.NakedObjectsSystem.IFixturesInstaller Fixtures {

get {

return new FixturesInstaller(

new CustomerFixture(),

new ProductFixture());

}

}

Where multiple fixtures are registered, as in the example code above, then the Install method on each fixture class will be called, in the order in which the fixture classes are registered. Each fixture class is installed within a separate transaction.

#### Determining when fixtures are to be installed

Assuming that you are working with a persistent object store then you don't want to install fixtures every time the application is run - otherwise you will end up with duplicates. However, you will typically want to install fixtures each time the object store is initialized - for example after a change to the model. For this reason, the EntityPersistorInstaller provides an IsInitializedCheck method. The following code illustrates its use:

protected override IObjectPersistorInstaller Persistor {

get {

var installer = new EntityPersistorInstaller();

System.Data.Entity.Database.SetInitializer(new DropDatabaseAndInstallFixtures<MyDbContext>());

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

installer.IsInitializedCheck = () => DropDatabaseAndInstallFixtures.IsInitialized;

return installer;

}

}

The IsInitializedCheck() method takes a boolean function, which in our example is defined on our database initializer, which inherits from any of the standard database initializer classes as shown below:

public class DropDatabaseAndInstallFixtures<T> : DropCreateDatabaseIfModelChanges<T>

where T : DbContext {

public DropDatabaseAndInstallFixtures() {

IsInitialized = true;

}

public static bool IsInitialized { get; set; }

protected override void Seed(T context) {

IsInitialized = false;

}

}

During the start-up process, after the EntityPersistor has been installed, Naked Objects will call the function assigned to IsInitializedCheck() and if it returns true, then it will run the Fixtures. If, upon start-up, the IsInitializedCheck() function returns false - because the database has not been initialized on this occasion, then the Fixtures are not run.

### Optional vs. Required

Naked Objects can be configured to run in two different ways:

1. All properties and all parameters are *required* unless marked up with the [Optionally] attribute. This approach is arguably the 'safer' approach when developing an application.

OR

1. All properties and all parameters are optional, unless marked up with the [Required] attribute. This approach is more compatible with the Entity Framework Code First approach - since the Required attribute is used to set the 'allow nulls' property on the database to false.

The configuration is determined by the OptionalByDefault property on the DotNetReflectorInstaller. By default, this is set to false, which represents the first configuration above. To specify the second configuration, then the OptionalByDefault property should be set to true, within the Reflector property on the Run class, as shown below:

protected override IReflectorInstaller Reflector {

get { return new DotNetReflectorInstaller { OptionalByDefault = true }; }

}

## 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 want to reference within domain model code.

### Recognised Value Types

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

* System.Boolean
* System.Byte
* System.Byte array (represents a 'blob' - typically an attached file)
* System.Character
* System.Decimal
* System.Double
* System.Enum (or, 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

See also: Section 2.4.6, “NakedObjects.Types”.

### 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 that represents 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> 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.

See Microsoft documentation.

#### 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. 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 Inline 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. In the user interface the user will only be able to Edit the Name when the Person is being edited also. (See also the Root attribute).

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; }

}

#### DataType

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

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 Hidden anyway, and should only be update-able through an action method.

#### DefaultValue

System.ComponentModel.DataAnnotations.DefaultValue

Allows you to specify, declaratively, the default value for any value input field.

See Microsoft documentation.

#### Description

System.ComponentModel.DataAnnotations.DisplayName

May be used as an alternative to the DescribedAs attribute in order to specify additional information that may be provided to the user (for example as a tool-tip).

See Microsoft documentation.

#### DisplayName

System.ComponentModel.DisplayName

May be used as an alternative to the NakedObjects.Named attribute to specify the name of a property or object that overrides the default name.

See Microsoft documentation.

#### MaxLength

System.ComponentModel.DataAnnotations.MaxLength

MaxLength is included with .NET 4.5, or, if running under .NET 4.0, with the Entity Framework package v 5.0.

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 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; }

}

#### MetadataType

System.ComponentModel.DataAnnotations.MetadataType

Specifies a separate class (type) that defines property-related metadata (attributes) for a domain class. This is most useful when working with partial classes that have been generated from an entity model. See [How to avoid accidentally over-writing your modifications to generated classe](#d0e530)s.

#### MinLength

System.ComponentModel.DataAnnotations.MinLength

This may be used in conjunction with an AutoComplete method. 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

See Microsoft documentation.

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 NakedObjects.RegEx attribute - to specify a regular expression that input text must satisfy on a string property. Note, however, that unlike the NakedObjects.RegEx attribute you cannot specify a formatting expression, nor explicitly control the case-sensitivity.

#### Required

See [Optional vs. Required](#_Optional_vs._Required).

Note that Required may also be used when operating Code First to ensure that the column in the generated database is Not Nullable. (At present this applies only to primitive properties, not toe reference properties).

#### 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 or 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.

At present, 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 (.NET) 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]...)

When such a method returns a reference the framework will attempt to display that object. If the method returns a null value then nothing will be displayed.

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. 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.
* 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.
* Clear: A complementary method used in conjunction with a property, the Clear method is called when the user (rather than the framework) clears a reference field, or blanks (so there is no entry) a value field. Changing a property from one value to another value, is deemed by the framework to be a 'clear field', immediately followed by a 'modify field'.
* 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 a null value then field or action remains enabled.
* 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. This is typically used for security reason, such as hiding a field once it is set up. Returning a true value makes the property or action invisible.
* 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 initialise an association (where an association is combination of references, such as a back link), or to trigger other behaviours such as updating a total.
* Validate: A complementary method used in conjunction with a property or an action

#### 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. For example, Persisted() is called after an object has been persisted.

* 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 in 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 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 *Persisting()* method. 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

* IconName: The IconName method provides an alternative mechanism (to the IconName attribute) for specifying the icon to be used by an object. It should be used wherever you wish to specify a different icon for different instances of a class, or according to the status of the instance. See [How to specify the icon for an object](#specify_icon).
* ToString: If no title attribute, or title method has been specified, then the framework will call the object's ToString method to get a title for the object.
* Title: If a single value property may be used to define a title for the object (and this approach is recommended where possible) then this is specified by means of a title attribute. If you wish to construct a title from several properties (value and or reference properties) and you do not wish to cache this combination as a property in its own right then you may specify the title in the form of a Title method that returns a string.

### NakedObjects.Attributes

NakedObjects.Attributes.dll includes a set of attributes that are recognised by the framework. See also Section 2.4.3, “Recognised .NET attributes”.

#### ActionOrder

Note: The recommended mechanism for specifying the order in which actions are listed to the user is MemberOrder. However, ActionOrder provides an alternative mechanism, in which the order is specified in one place in the class, with the added advantage (currently) that you can easily specify groupings (which may be rendered by the framework as sub-menus). However, ActionOrder is more 'brittle' to change: if you change the name of an existing action you will need to ensure that the corresponding name within the ActionOrder attribute is also changed.

The syntax is:

[ActionOrder("<comma separated list of action names>")]

For example:

[ActionOrder("PlaceOrder, CheckCredit")]

public class Customer

{

public Order PlaceOrder() {...}

public void CheckCredit() {...}

}

The action names are not case sensitive within the ActionOrder list.

Actions can be grouped together by surrounding the group with brackets, and prefixing the group with name and colon. This information may be used by the viewing mechanism to render actions into sub-menus. For example:

[ActionOrder("(Account Management: PlaceOrder, CheckCredit), (Personal Details: ChangeOfAddress, AddEmail)")]

public class Customer

{

public Order PlaceNewOrder() {...}

public void CheckCredit() {...}

public Address ChangeOfAddress() {...}

public void AddEmail(string emailAddress) {...}

...

}

#### 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).

#### 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).

#### 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 viewer will use this information to render all the instances of this class available to the user in a convenient form, such as a drop-down list.

#### Debug

The Debug attribute marks an action method as available in debug mode only, and so will not normally be displayed by the user interface.

#### DescribedAs

Subsequent to Naked Objects' introduction of the DescribedAs attribute, Microsoft has introduced its own Description attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of DescribedAs.

The DescribedAs attribute is used to provide a short description of something that features on the user interface. How this description is used will depend upon the viewing mechanism - but it may be thought of as being like a 'tool tip'. Descriptions may be provided for objects, members (properties, collections and actions), and for individual parameters within an action method. DescribedAs therefore works in a very similar manner to Named.

To provide a description for an object, use the DescribedAs attribute immediately before the declaration of that object class. For example:

[DescribedAs("A Customer who may have originally become known to us via the marketing system or who may have contacted us directly.")]

public class ProspectiveSale {...}

Any member (property, collection or action) may also provide a description. To specify this description, use the DescribedAs attribute immediately before the declaration of that member. For example:

public class Customer

{

[DescribedAs("The name that the customer has indicated that they wish to be addressed as (eg Johnny rather than Jonathan)")]

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

}

To provide a description for an individual action parameter, use the DescribedAs attribute in-line. For example:

public class Customer

{

public Order PlaceOrder(

Product product,

[DescribedAs("The quantity of the product being ordered")]

int quantity)

{...}

}

#### Disabled

The Disabled attribute means that the member cannot be used in any instance of the class. When applied to the property it means that the user may not modify the value of that property (though it may still be modified programmatically). When applied to an action method, it means that the user cannot invoke that method. For example:

public class Customer

{

[Disabled]

public void AssessCreditWorthiness() {...}

[Disabled]

public virtual Money InitialCreditRating() {...}

}

Note that if an action is marked as Disabled, it will be shown on the user interface but cannot ever be invoked. One possible reason to do this is during prototyping, to indicate an action that is still to be developed. If a method is intended for programmatic use, but not intended ever to be invoked directly by a user, then it should be marked as Hidden instead. This attribute can also take a single parameter indicating when it is to be hidden, for example the following code would disable the action until the object has been saved.

public class Customer

{

[Disabled(WhenTo.UntilPersisted)]

public void AssessCreditWorthiness() {...}

}

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 disabled.

#### Eagerly

Eagerly may be applied to any action or property, including a collection, within a class. The attribute may also be applied at class level - indicating that all properties and collections within that class should be eagerly rendered.

If the model is being run with the Restful Objects Server this specifies that the details and contents ('value') of the property are to be rendered in-line - containing the same JSON representation as if the client had followed the link(s) directly.

If the model is being run with Naked Objects MVC, this attribute causes the associated object or collection to be rendered in-line within the HTML. In the case of a reference property, the resulting HTML is the same as if the user had viewed the object and then expanded the referenced object in-line; in the case of a collection it is as if the user had viewed the object and then selected the table view on that collection. If applied to an action that returns a collection, then the result will be rendered as a table rather than a list. (This may be useful combined with the use of 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;}

In future releases we expect that the Do Enum will offer more options.

At present, Eagerly is a mechanism to specify the 'eager rendering' of information passed from the server to the client - it does not have any bearing on the loading of information from the database to the server ('eager loading'). Again, this is likely to change in a future release.

#### ExcludeFromFindMenu

Currently, this attribute is relevant only when working with Naked Objects MVC. It may be applied to an action on a service to prevent that action from being rendered within the Find menu for a property/parameter of that type.

#### Executed

When running a Naked Objects MVC application using the Ajax functionality, Validate methods will normally be called on the server as the user is typing in a value. If the Validate method contains very complex logic, then this could result in a significant performance overhead. Server-side validation using Ajax may therefore be suppressed on a field-by-field basis, by adding the attribute [Executed(Ajax.Disabled)].

#### FieldOrder

Note: The recommended mechanism for specifying the order in which fields are listed to the user is MemberOrder. FieldOrder provides an alternative mechanism, in which the order is specified in one place in the class. However, FieldOrder is more 'brittle' to change: if you change the name of an existing property you will need to ensure that the corresponding name within the FieldOrder attribute is also changed.

The FieldOrder attribute should be provided with a comma-separated list of field names, which are not case sensitive. For example:

[FieldOrder("Name, Address, DateOfBirth, RecentOrders")]

public class Customer

{

Public virtual Date DateOfBirth() { get; set; }

Public virtual List<Order> RecentOrders() { get; set; }

Public virtual string Address() { get; set; }

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

}

#### Hidden

The Hidden attribute indicates that the member (property, collection or action) to which it is applied should never be visible to the user For example:

public class Customer

{

[Hidden]

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

[Hidden]

public void UpdateStatus() {...}

}

This attribute can also take 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. By default the annotated property or action is always hidden.

#### IconName

The IconName attribute applies at class level, and provides the simplest mechanism for specifying the name of the icon to be used for that type. See [How to specify the icon for an object](#specify_icon).

[IconName("person.gif")]

public class Customer {...}

#### Idempotent

Applies to an action. Used to indicate to a client that the action is 'idempotent' - invoking it more than once in succession (with the same arguments) has the same effect as invoking it once. 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 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 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 allows you to apply standard .NET masks to format the presentation of DateTime or numeric value properties. 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.

A very handy 'cheat sheet' on using .NET masks may be downloaded from: http://www.cheat-sheets.org/saved-copy/msnet-formatting-strings.pdf.

Note that the mask applies only to the presentation of the value - it does not have any impact on the parsing of the input. For Naked Objects MVC, the input parsing is determined by the language settings on the user's browser. (Though it is always better to avoid ambiguity by using the JQuery date picker for date entry if possible.)

If you are writing an application that may be used by users with different language/locale settings, then you should avoid using very specific masks that could lead to an existing value being mis-interpreted when an edited object is saved. Thus, while [Mask("d")] is safe, [Mask("dd-MM-yy")] is not safe - because a user with US settings could interpret certain existing dates as "MM-dd-yy".

#### MemberOrder

MemberOrder is the recommended mechanism for specifying the order in which fields and/or actions are presented to the user. (ActionOrder and FieldOrder provide alternative mechanisms).

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 the sequence specifier String, using the standard String comparator.

The simplest convention is to use numbers - 1, 2, 3 - though it is a better idea to leave gaps in the numbers - 10, 20, 30 perhaps - such that a new member may be added without having to edit existing numbers. A useful alternative is to adopt the 'dot-decimal' notation - 1, 1.1, 1.2, 2, 3, 5.1.1, 5.2.2, 5.2, 5.3 - 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;}

}

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.)

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.

Note that certain styles of user interface will lay out an object's properties and its collections separately, in which case the relative member order of properties and collections will be evaluated separately. However, since other styles of user interface may interleave properties and collections, it is safer to assume the latter.

When applied to actions, it is also possible to specify a sub-menu for the action:

[MemberOrder(Name="Sub Menu A", Sequence="10")]

public virtual void MyAction() {…}

where Name is the name of the desired sub-menu, and Sequence (which must be specified as a string rather) specifies the order *within the sub-menu.* (It is not currently possible to specify where the sub-menu appears in relation to other actions or sub-menus)

#### 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, Width = 40)]

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) then the field will be rendered with vertical scrolling.

In NakedObjects MVC, the Width parameter only has an effect if the width of the textarea is not overridden by the .CSS.

#### NakedObjectsIgnore

This attribute may be applied to a property or method that contains within its signature a type that you wish Naked Objects to ignore - in other words not to introspect. Such types cannot then be displayed on the user interface as Naked Objects, nor persisted via the normal mechanism. The typical use of this capability is where you are incorporating a third party class library that is used to provide some specific functionality, not to provide domain objects. Marking any such references with NakedObjectsIgnore may be motivated by a desire to minimise unnecessary introspection at start-up, or because the third-party library uses constructs that cannot be processed by the Naked Objects introspector and would thus throw errors.

#### Named

Subsequent to Naked Objects' introduction of the Named attribute, Microsoft has introduced its own DisplayName attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of 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.

#### NotContributedAction

Indicates that an action on a service should not be treated as a contributed action: in other words it should not be added to the object actions for any object types that appear in its parameter list.

#### NotPersisted

This attribute indicates that a class is not to be persisted. The user may still edit such objects and the Save button will confirm those edits - but the object will never be persisted.

This attribute will work with Entity Framework in Model First or Database First mode, but not Code First. The Microsoft Code First team has said that they intend to introduce such an attribute called, perhaps, StoreIgnore in the near future. When that appears, we will obsolete our own NotPersisted attribute in favour of the Microsoft one.

This is typically used on classes that are defined solely for the purpose of inputting data or presenting data. For example:

[NotPersisted]

public class CustomerSummary {...}

The NotPersisted attribute may also be applied at the level of an individual property, as shown below. (If the property is ReadOnly (has only a get method in C#) then it will not be persisted anyway. The NotPersisted attribute is necessary only if the property needs to have a set method for other purposes.)

public class Order {

[NotPersisted]

Public virtual Order PreviousOrder() { get; set; }

}

Note that if a persisted class has a property of a type that is marked NotPersisted then the property itself must also be marked NotPersisted - otherwise an error will arise.

#### Optionally

See [Optional vs. Required](#_Optional_vs._Required).

#### 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. If the value is set to zero, no paging will occur and the full set of objects returned.

#### 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 mi 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 Naked Objects MVC these hints are rendered as class(es) in the <div> (or other Html element) that most closely corresponds to the location of the hint. For example, the Html for the Text field above will be rendered thus:

<div class="nof-property rich-text extra-wide" id="Foo-Text>...</div>

This information may be used by adding a custom style sheet and/or custom JavaScript. (The generic Naked Objects MVC user interface does not recognise any specific hints.)

In Restful Objects, 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"

#### ProgramPersistableOnly

This attribute, when applied at class level, indicates that transient instances of a class may only be persisted programmatically - not directly by the user. This is typically used where you wish to create one transient object within another transient object (sometimes referred to as the 'aggregation' pattern) and you do not wish the user to be able to persist the child object separately from the parent object. By marking the child object with the ProgramPersistableOnly attribute the user cannot persist it directly (hitting the Save button while that object is transient confirms the edits but does not persist the object). However assuming that the Parent object holds a reference to that Child object, then the child will be persisted automatically when the parent is persisted.

#### Prototype

If a method is marked up with the Prototype attribute, then it will be made available to the user as an action only if the system is configured to show prototype actions. This is useful for writing methods that can be helpful when exploring a prototype, but would be unwanted, or potentially dangerous on a deployed system, for example a repository action that returns all instances of a given type.

[Prototype()]

public List<Customer> AllCustomers() {...}

To show prototype actions, add following code to your Run class:

protected override bool ShowPrototypeActions {

get { return true; }

}

#### QueryOnly

Applies to an action. Used to indicate to a client that the action is 'side-effect free' (does not change the persistent state of the system). Note that this is purely an indicator: adding the attribute does not prevent a method from changing the state. Application developers should therefore take considerable care to ensure that this attribute is applied correctly. (See also the Idempotent attribute).

#### RegEx

Subsequent to Naked Objects' introduction of the RegEx attribute, Microsoft has introduced its own RegularExpression attribute. As the latter is now recognised by Naked Objects, we recommend that you use it in place of 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>, Message = <string>, CaseSensitive = <True|False>)]

The first parameter is required; the message property is optional and defines a message to return to the user if the text entered does not match the RegEx string; caseSensitive is optional, defaulting to false. 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]+$", Message = "Not a valid email address")]

public virtual string Email { get; set; }

}

When applying the RegEx attribute to a value parameter within an action method, the attribute should precede that parameter:

public class Organisation{

public void NewContact(

string contactName,

[RegEx(Validation = @"^[\-\w\.]+@[\-\w\.]+\.[A-Za-z]+$",   
 Message = "Not a valid email address")]

string email) {...}

}

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 the example used in Section 2.4.3.2, “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, "Description", "DueDate", "onfidential"]

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

The first, boolean, parameter of the attribute determined whether or not the title of each object will be rendered as the first column of the table - since this title also acts as a link to a standalone view of that object then the parameter will typically be set to true if you want the user to be able to navigate to an object in the collection; it is typically set to false where the table view shows all the information the user needs and there is nothing to be gained from navigating the individual object. 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.
* [TableView(true)] will result in a single-column table with just the title of the object, equivalent to a 'list view'.
* If an object or collection is displayed using a custom view that specifies the columns to be displayed then the latter will take precedence over the TableView annotation (if one exists).

#### 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 if one existed). Normally, this should be applied to a property containing a value type such as String, or Date. However, 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

The TypicalLength attribute indicates the typical length of a String property or parameter in an action. This may be used by the viewing mechanism to determine the space that should be given to that property or parameter in the appropriate view. For example, in a table view, this will determine the default width for that column.

public class Customer {

[StringLength(30)] [TypicalLength(15)]

Public virtual string FirstName() { get; set; }

}

If the typical length is the same as the StringLength then there is no need to specify TypicalLength as well. If the value specified is zero or negative then it will be ignored.

In NakedObjects MVC, TypicalLength only has an effect if the width of the input is not overridden by the .CSS.

#### 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.

### NakedObjects.Types

The NakedObjects.Types assembly provides a number of interfaces and classes that are explicitly recognised by the Naked Objects framework, and that intended to 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 and exceptions raised in application code.
* IDomainObjectContainer. See [IDomainObjectContainer methods](#IDomainObjectContainer)”.
* IViewModel and IViewModelEdit. See [View Model](#_View_Model).

#### NakedObjects.Async

* IAsyncService. Interface representation of the AsyncService class (in NakedObjects.Asynch.dll) - for injection into domain code.

#### NakedObjects.Audit

* IAuditor. Allows domain programmers to define an auditing service (registered via the AuditorInstaller on the Run class).
* INamespaceAuditor. Specific sub-type of IAuditor, where the methods will only be called in relation to types that fall within the specified namespace.

#### NakedObjects.Redirect

* IRedirectedObject. (Intended for use with Restful Objects) Implemented by a 'stub' class that acts as a proxy for an object managed on another server.
* IRedirectedService. (Intended for use with Restful Objects). Implemented by a 'stub' class that acts as proxy to a service implemented on another server. Note that, unlike IRedirectedObject, this defines functions, not properties.

#### NakedObjects.Security

* INamespaceAuthorizer. An implementation of this interface provides authorization for a single fully-qualified type, or for any types within a namespace. The implementation should be registered via the TypeAuthorizerInstaller in the Run class. 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 Section 2.2.2, “View Model”.
* IHasGuid. This interface is intended for use with 'Polymorphic Associations' 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. See [How to handle associations that are defined by an interface rather than a class](#interface_association).
* 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.
* TitleBuilder and NewTitleBuilder. Helper objects for constructing a title string on an object. (The latter provides a slightly different syntax that offers a finer level of control).

#### 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).
* SimpleRepository. See [SimpleRepository](#_SimpleRepository).

#### NakedObjects.Utils

* AttributeUtils. Utility methods for getting custom attributes declared on domain types, members or method parameters.
* 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.

# Adding behaviour to your domain objects - a how-to guide

This section is concerned with adding to, or modifying, the behaviour of those domain objects, in order to build a richer application. It is presented in the form of a 'How-to Guide': we have tried to cover all the common questions that programmers raise about adding behaviour.

## 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 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.

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 a fixture class.
* Repository Creates a new repository (and/or factory) class inheriting from AbstractFactoryAndRepository.
* XAT creates a new class to contain XATs.
* Default\_T4Template Adds a T4 template that can translate a .edmx file into domain classes suitable for use with Naked Objects. Intended for use when creating a model from an existing database.
* **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 Section 2.3.1.6, “How to insert behaviour into the object life cycle”.

### 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 Section 2.3.1.6, “How to insert behaviour into the object life cycle”.

### 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 Section 2.3.1.6, “How to insert behaviour into the object life cycle”.

### 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 NotPersisted attribute. (See also the ProgramPersistableOnly attribute).

### 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.TitleBuilder object to build the title. TitleBuilder 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()

{

TitleBuilder t = new TitleBuilder();

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 TitleBuilder.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 specify the icon for an object

Naked Objects allows you to associate an icon with an object, either based on the object's type (using the the IconName attribute ) or specific to the instance (using an IconName() method) - explained below. Naked Objects can accept most common image types as an icon, including .gif, .png, and .ico.

For Naked Objects MVC, the icons should be included in the Images folder of the Run project - as with any ASP.NET MVC project. If no icon name is specified for an object, then Naked Objects MVC will specify the icon in the generated HTML as default.png.

The the IconName attribute is applied at class level as shown below:

[IconName("person.png")]

public class Customer {...}

The IconName method allows you to specify an individual icon for each instance of a class, if you need to. For example, an instance of Product could use a photograph of the product as an icon, using:

public string IconName() {

return ProductName() + "-photograph.png";

}

Use the IconName snippet (shortcut icon) for creating this method.

It is also possible to vary the icon according to the status of the object:

public string IconName() {

return "Order-" + Status() + ".png";

}

As shown in these examples, we recommend that you include the file extension with the icon name. If no extension is specified, the system will look for a .png file with that name.

### 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)

See [Optional vs. Required](#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 Section 2.3.3.18, “Enums”.

See also Section 2.3.3.9, “How to specify auto-complete for a 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 value properties (e.g. a string or int 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.

### 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 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. (If you are using Naked Objects MVC then 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" };

}

}

[Hidden]

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

[Hidden]

public virtual string AttName { get; set; }

[Hidden]

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);

}

}

[Hidden]

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

[Hidden]

public virtual string PhotoName {get; set;}

[Hidden]

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).

## 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 Section 2.4.2, “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 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 Naked Objects MVC 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

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.

### 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 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 attribute. Or if none of the methods on your service are intended to be user actions, then register that service within the SystemServices property on the Run class.

### 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

See [Optional vs. Required](#_Optional_vs._Required).

### 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, the you can create 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 Section 2.3.3.18, “Enums”.

See also Section 2.3.5.4.6, “How to specify auto-complete for a parameter”.

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. There is a second syntax also recognised:

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

However, this is no longer recommended as there is a potential ambiguity if the object on which the action exists happens to have a Property that matches the type and name of the parameter passed in.

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 value parameters (e.g. a string or int ), 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.

### 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 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.

#### Disabling an action for certain users or roles

See 'Properties: Disabling a property for specific users or roles'. The same technique can be applied to actions. However, the caveats apply.

### 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 throw an application exception

See ???

### 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.

## Advanced Entity Framework techniques

This section provides a few notes on using more advanced features of the Entity Framework with Naked Objects for .NET.

### 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.)

### 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 (see Section 2.1.1, “Creating a Model project from scratch” ) 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 avoid losing your code modifications

If you are working from an existing database you will be auto-generating classes by means of the Transformation Template. However, anytime you re-run this template, the previously-generated classes will be over-written. If you have added any code to those generated classes, this code will be lost.

The generated code is all in the form of partial classes - so the best way to eliminate the risk of losing your additional code is to write it into a separate partial class. This means that you can leave the generated classes unmodified, and add all your behaviour into a separate partial class with the same name. Although this makes the code slightly less easy to read (because the capabilities of each class are split across two or more files), it does mean that you won't have to worry about regenerating the model.

However, partial classes have one limitation that is, unfortunately, very relevant to working with the Entity Framework: which is that properties can only be declared once within partial classes. This means that you can't have a property that is defined in a partial class that has been generated by the transformation template, and then specify attributes for that property in a separate partial class. This is a severe limitation that impacts anyone using Entity Framework in model-first mode, and is recognised as such by Microsoft. The long term solution lies in a change to the definition of the .NET languages.

Meantime, Microsoft has come up with a work-around solution which is the use of an 'associated metadata class' - sometimes informally referred to as a 'buddy class'. It is well described in this: blog entry. Using this pattern, each domain class is split into three parts:

1. A partial class auto-generated by the transformation template, which contains all the (persisted) properties.

A partial class manually generated by the programmer, which contains methods and any non-persisted properties. This partial class is marked up with a MetadataType attribute, which specifies the third item...

1. A standalone class (usually having the same name as the domain class, but with the suffix: \_Metadata) that defines the same set of properties as the domain class, with attributes added to them where needed. This class is sometimes informally referred to as the 'buddy class'. The programmer needs to take responsibility to ensure that the list of properties remains fully consistent between the domain class and this class - though any discrepancy will throw a clear error at runtime.

The second two elements could readily be combined into a single file.

At runtime, the Naked Objects framework will, for a given property on a domain class, recognise any attributes that are defined either on the property within the domain class, or on the corresponding property in the run class. Naked Objects provides an item template - 'Partial Domain Object With Buddy' - which creates a partial class and a corresponding metadata (buddy) class in one file. You should create this new item within a separate folder, to avoid over-writing the generated partial class.

The following example code shows the example of a Product object implemented in the form of the three elements listed above:

// This partial class has been generated automatically from the entity model

// and is not touched by the application programmer.

// It contains all the properties for Product.

public partial class Product {

public virtual int Id { get; set; }

public virtual string Name { get; set; }

public virtual int UnitsInStock { get; set; }

}

//Use this partial class to define actions for the Class1 class.

//Also for methods to enrich the behaviour of properties - such as Validate,

//Default, or Choices and for class-level attributes such as <IconName>.

[MetadataType(typeof(Product\_Metadata)]

[IconName("Package.png")]

public partial class Product {

public void RecordReceiptOfNewStock(int Number) {...}

public string ValidateName(string value) {...}

}

//This is the so-called 'buddy class' for Class1. It should have the same

//properties as the auto-generated partial Class1 class, to which attributes

//property-level attributes such as <Hidden> may be added.

public class Product\_Metadata {

[Hidden]

public virtual int Id { get; set; }

[StringLength(25)]

public virtual string Name { get; set; }

[Disabled]

public int UnitsInStock { get; set; }

}

### How to implement complex types

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).

When working Code First it is necessary only to annotate the complex type class with the ComplexType attribute. Alternatively, you may specify a complex type by means of the Code First Fluent API (see Section 2.1.1.1, “Overriding the default database schema generation”).

### 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 EntityPersistorInstaller. See [Creating a Model project from scratch](#_Creating_a_Model).

### 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 'polymorphic association' and the 'result interface association'

The polymorphic association is used where you have an interface that represents a role that is potentially played by multiple objects (and where you cannot assume that those objects all inherit from a common superclass).

The Naked Objects host site on Codeplex has a document (under 'Patterns and Practices') that shows how to implement the polymorphic association pattern using code snippets and helper classes now provided by the Naked Objects programming library.

The second pattern applies to 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;

}

}

}

### How to work with multiple database contexts

Naked Objects provides very good support for working with multiple contexts, including explicitly-coded contexts (as used in Code First mode) or implicit contexts (as used when working with entity models defined in .edmx files), or a combination of the two, as shown below:

protected override IObjectPersistorInstaller Persistor {

get {

var p = new EntityPersistorInstaller();

p.UsingEdmxContext("ModelA"); //Has corresponding .edmx file

p.UsingEdmxContext("ModelB"); //Has corresponding .edmx file

p.UsingCodeFirstContext(() => new ModelCContext());

p.UsingCodeFirstContext(() => new ModelDContext());

return p;

}

}

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:

protected override IObjectPersistorInstaller Persistor {

get {

var p = new EntityPersistorInstaller();

p.UsingEdmxContext("ModelA").AssociateTypes(ModelATypes); //Has corresponding .edmx file

...

p.UsingCodeFirstContext(() => new ModelCContext()).AssociateTypes(ModelCContext.ModelCTypes);

...

return p;

}

}

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 EntityPersistorInstaller. 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:

protected override IObjectPersistorInstaller Persistor {

get {

var installer = new EntityPersistorInstaller();

installer.UsingEdmxContext("Model").AssociateTypes(AdventureWorksTypes);

installer.SpecifyTypesNotAssociatedWithAnyContext(() => new[] { typeof(AWDomainObject) });

return installer;

}

}

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 EntityPersistorInstaller 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).

## Customising the MVC User Interface

There are three ways that you can customise the generic HTML user interface that is created by Naked Objects MVC automatically:

### Customising the CSS

The generic user interface created by Naked Objects MVC relies on CSS (Cascading Style Sheets) for all of its styling. From within your browser you can temporarily switch the styling off - the result is not pretty but it is fully functional. (You will see, for example, that the drop-down menus are created entirely through styling rules. The only use of JavaScript is for the pop-up calendar helper, which will simply not show if you have JavaScript turned off).

You can customise the generic user interface to a surprisingly high degree just by modifying the .css file(s). You can, for example, apply styling changes to individual classes of domain object, individual properties within an object, individual actions, or individual parameters within an action dialog, and so on. This is because the HTML generated by Naked Objects MVC makes extensive use of the class and id attributes. Consider the following view of a Store object from the AdventureWorks demo:



The HTML for the Sales Person property is shown below (formatted for clarity):

<div class="Property" id="Store-SalesPerson">

<label>Sales Person:</label>

<div class="Object" title="">

<img alt="Sales Person" src="/Content/dog.gif" />

<a ref="/Store/Details/AdventureWorksModel.SalesPerson%3bAdventureWorksModel.  
 SalesPerson%3b1%3bSystem.Int32%3b290%3bFalse%3b%3b">Lynn Tsoflias</a>

</div>

</div>

We can use the class and id to specify styling just for this property, for example, by adding the following to the Site.css (or to another .css file that is imported into site.css):

.Property#Store-SalesPerson { color: Red; }

to get the following effect:



This is a trivial example, but you can use similar techniques to control the whole layout and visual presentation of individual objects and actions.

Writing .css can be a challenging exercise and you are advised not to try it until you have gained a thorough grounding in the technology and techniques of .css, which is outside the scope of this manual. However, customisation entirely through .css offers these huge advantages:

* You can modify the .css files while the solution is running to see the effect (you just need to hit Refresh on your browser each time).
* Because changing the CSS does not impact the functionality of the application, you can manage the styling of the application as a completely independent process - for example delegating it to a specialised designer. The application development is not then dependent upon the styling process.

### Custom Views

If you are unable to achieve the customisation that you need entirely through css, then the next option is to create custom views. A typical example of this is where you wish to display more on a page than exists for the standard generic view. For example:

* If an Order object contains a collection of OrderLine objects, then the default view will show this in summary form, requiring the user to click on the List or Table button to see more (this 'lazy loading' approach allows the Order to be retrieved faster). However you might wish to have the collection of order lines rendered as a table even when the order is first retrieved.
* The Order will probably also have an action AddNewLine available on its action menu, and which returns a dialog asking for the Product and Quantity, say. With a customised view you could arrange to have this action dialog exist on the same page as the Order - but only if the order status showed that it was not already shipped.

Writing a new view is quite straightforward. The easiest approach is to start from one of the default views that are used by a Naked Objects MVC project: you will find them under: Views > Shared. These views follow standard ASP.NET MVC conventions. For example, by default a single object will be displayed using ObjectView.aspx.

To create a customised view for the Order object, just create a new folder within Views called Order, copy the ObjectView.aspx file into this folder, keeping the same name for the file - and then customise the new view. Naked Objects MVC will automatically detect, and use, this new view whenever it is displaying an Order. If you subsequently remove or rename that file or folder then Naked Objects MVC will revert to using the default ObjectView.aspx file in Views > Shared.

Similarly, you can create a customised view for a collection of Order objects by copying StandaloneTable.aspx from the Shared folder to the Order folder, and then modifying it.

When writing custom pages and using AJAX you should be aware that only the content on the page within the div#main tag will be updated from the AJAX request. This means that any custom content outside of div#main (such as scripts and style tags added into the custom page) will not be copied into the page. For this reason we recommend that scripts and style tags are not directly added to a custom page but are coded into separate .css or .js files. These may then be referenced from the master pages. Scripts should adopt the 'unobtrusive' style of Javascript programming and use JQuery events and binding. If you do add scripts and style tags into a custom page be aware that you may need to do some further custom coding to have those scripts execute.

A set of Naked Objects HTML helper methods make it really easy to create custom views - and keep the resulting code really small. As with the standard HTML helper methods (that form part of ASP.NET MVC) these take the form extension methods on the type HtmlHelper. They are defined on two classes: NofHtmlHelper and CustomHtmlHelper (both in the namespace NakedObjects.Web.Mvc.Html). Those defined on NofHtmlHelper are used within the standard shared views in Naked Objects MVC but these may also be used in your own custom views; those defined on CustomHtmlHelper provide additional capabilities intended specifically for custom views.

The following example shows a custom page to view a Workflow object using several of the helper methods:

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.WithServices.Master" Inherits="System.Web.Mvc.ViewPage<MyApp.Model.Workflow>" %>

<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>

<%@ Import Namespace="MyApp.Model.Workflow"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

<%: Html.ObjectTitle(Model)%>

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<div class="ObjectView" id="<%: Html.ObjectTypeAsCssId(Model) %>">

<%: Html.Object(Model)%>

<%: Html.Menu(Model)%>

<%: Html.UserMessages() %>

<%if (Html.ObjectHasVisibleFields(Model)) {%>

<%: Html.PropertyListWithout(Model, x=> x.SubWorkflows)%>

<div id="Process-PropertyList1" class="PropertyList">

<div class="Property" id="Workflow-SubWorkflows"><label>Sub Workflows:</label>

<%: Html.CollectionTableWith(Model.SubWorkflows, "Order", "WorkflowName", "Status", "Priority")%>

</div>

</div>

<%: Html.GenericAction("Edit", "EditObject", Model)%>

<%}%>

</div>

</asp:Content>

Note the following:

* Inherits="..." This is what is known in ASP.NET MVC terminology as a 'strongly-typed' view - all references to Model will be to an object of type MyApp.Model.Workflow.
* <%: Html.ObjectTitle(Model)%> specifies the title of the page to be the title of the (Workflow) object that we are viewing.
* <div class="ObjectView" id="<%: Html.ObjectTypeAsCssId(Model) %>"> Specifying the id of the div using this helper method allows the .css to style this object view specifically, should we want that.
* <%: Html.Object(Model)%> adds the icon and the title for the object
* <%: Html.Menu(Model)%> adds the menu of actions for this (Workflow) object.
* <%: Html.UserMessages() %> specifies that this is where any user messages, such as validation errors, are to be rendered.
* <%: Html.PropertyListWithout(Model, x=> x.SubWorkflows)%> generates a list of all the properties of the object, except for the SubWorkflows property (which is rendered separately, below). Note that we could have written this more simply as <% Html.PropertyList(Model, "SubWorkflows") %>. The advantage of the slightly more complex syntax, which uses a Lambda expression, is that it is type-safe: if the SubWorkflows is subsequently renamed in the domain object to, say, SubsidiaryWorkflows, then this will be automatically renamed here; this would not be so if the name of the property is defined by a literal string.
* The next three lines render a view of the objects SubWorkflows collection, showing only the columns Order,WorkflowName,Status and Priority. The reason for manually adding the two divs around this is simply to pick up the styling for collections from the standard .css - it could have been omitted and styled separately.
* <%: Html.GenericAction("Edit", "EditObject", Model)%> provide the generic Edit action (rendered as a button by the standard .css).

The following is an incomplete list, showing just some of the helper methods that you may like to use in creating your own custom views:

#### ServiceExtensions

Services - renders the complete set of service menus available to the user.

SingleServiceMenu - renders a single service menu.

SystemMenu - adds a menu that provides certain system level actions. This is normally used during prototype development only, as the actions are not typically relevant (or safe) for end-users.

#### ObjectExtensions

Object - renders the icon and title for a specified object

IconName - displays the name of the icon for a specified object

#### PropertyExtensions

ObjectPropertyView - renders a complete view of a single specified object, with an Edit button.

ObjectPropertyEdit - renders a complete view of a single specified object in Edit mode, with a Save button.

PropertyList - displays all of an object's properties

PropertyListWithout - displays all of an object's properties, except those specified.

PropertyListWith - displays a specified list of properties of an object.

PropertyListEdit - As for PropertyList but rendered in Edit mode.

PropertyListEditWithout - As for PropertyListWithout but rendered in Edit mode.

PropertyListEditWith - As for PropertyListWith but rendered in Edit mode.

ObjectHasVisibleFields - Returns true if an object has any properties that could be displayed to this user.

Contents - displays the contents of a specified object property

Name - displays the name of a specified object property

Description - displays the description (if one has been provided in the Model) of a specified object property

TypeName - displays the object type of the property

#### ActionExtensions

Menu - Renders all of the actions on an object (that are available to the user) as a menu

ObjectAction - Renders a specified action from an object in the form of a button.

ObjectActionAsDialog - Renders a specified action from an object in the form of a dialog.

ParameterList - renders the list of parameters for a specified action.

ActionDialogId - renders the ID for an action (to allow it to be invoked by the Naked Objects framework.

ActionName - renders the (formatted) name of the action.

ControllerAction - renders one of the generic framework actions buttons such as Edit, Select, or Remove.

UserMessages - renders the message resulting from validation errors (on an action dialog or attempting to save an edited object)

#### CollectionExtensions

CollectionTable - renders a specified collection as a table view showing all properties as columns

CollectionTableWith - renders a specified collection as a table view, including only the specified properties as columns

CollectionTableWithout - renders a specified collection as a table view, including all properties, except those specified, as columns

#### Helpers that apply to objects, properties and/or parameters

Contents - renders the contents of a specified object property or action dialog parameter.

TypeName - renders the name of the object type (class) for a specified object, object property or action dialog parameter.

Name - renders the name (label) for a specified object property or action dialog parameter.

Description - renders the description (tool-tip text)for a specified object property or action dialog parameter.

### Custom controllers

The third level of customisation is to write your own custom controllers. Examples of where this would be needed include:

* Where you have more than one view for an object (or collection of objects) and you wish to use different views in different contexts. (However, you should first ask yourself if it would be possible to achieve the same effect by having a single view that had conditional statements built into it - as this would require no new controller).
* Where you wish to provide a view that contains than one domain object (other than a collection of domain objects, which is handled already), or a domain object plus some additional data. This can be done either by feeding additional objects into the ViewData associated with a view, or by creating a 'View Model'. These are both considered to be fully legitimate within ASP.NET MVC. However, we caution against over-use of these patterns. If you are having to create a lot of View Model objects, for example, this might suggest that your underlying domain model is not a good match to the business domain.
* Where you wish to control, explicitly, the flow of the application. By default, a Naked Objects MVC application allows the user to perform any legitimate action in any context (while still enforcing necessary rules). This is a very 'empowering' style of user interface and gives the user a great deal of control. But this style of user interface has a learning curve - the user is not told what to do next. It is appropriate for many internal-facing applications for a business, but very seldom appropriate for external-facing applications, where a more 'narrow', step by step, style of interface is called for.

Custom controllers are written using standard ASP.NET MVC patterns. The more custom controllers you write, the more your application starts to resemble a conventional ASP.NET MVC application. So add them only as you are sure that you need them. A recommended strategy is:

1. Start by creating an internal 'power-user' application using default Naked Objects patterns and with the minimum possible use of custom controllers (you might well find that none are needed).
2. Then add one or more narrowly-scripted applications for external use, written using conventional ASP.NET MVC patterns, but leveraging the Naked Objects helper methods in the views and/or controllers.

To start to write custom controllers in Naked Objects MVC, you must be familiar with how to write conventional ASP.NET MVC applications. Books, videos and other training materials are readily available from Microsoft and other sources. If you do not have familiarity with writing MVC applications, you will have difficulty following the example below, let alone write your own.

#### Example

This example demonstrates a Naked Objects MVC application with a highly-constrained user interface that makes use of (amongst other things) a custom controller. It is taken from the simple e-commerce application was used on the Naked Objects website prior to the framework being open-sourced! Let's first of all walk through the finished application from a user's viewpoint. The starting point is a regular HTML page on the website:



Clicking on the top-most Click to purchase link takes us to a page where we specify the number of licenses to purchase and customer details:



Having entered at least all the mandatory fields and hitting save, we are presented with a confirmation screen:



Pressing Confirm & Proceed to Payment will take you to on the PayPal website; Edit takes you back to the previous screen to amend details.

Let's now look at the implementation.

If we look at the HTML source for the first screen, we can see that the three Click to purchase links take the form:

<a href="http://salesnakdobjects.cloudapp.net/OrderEntry/  
 CreateNewOrderEntry?productCode=00020&currency=GBP">Click to purchase<a>

The URL of the link gives away that the application is actually running in the cloud, under Microsoft's Azure. If we look at the RegisterRoutes method in the application's Global.asax.cs file ...

public static void RegisterRoutes(RouteCollection routes) {

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

"Product Details",

"Product/Details",

new {controller = "Generic", action = "Details"}

);

routes.MapRoute(

"Country Details",

"Country/Details",

new {controller = "Generic", action = "Details"}

);

routes.MapRoute(

"EditObject",

"OrderEntry/EditObject",

new {controller = "Generic", action = "EditObject"}

);

routes.MapRoute(

"Edit",

"OrderEntry/Edit",

new {controller = "Generic", action = "Edit"}

);

routes.MapRoute(

"Default",

"{controller}/{action}",

new {controller = "OrderEntry", action = "Index"}

);

}

... we can see that the URL ...OrderEntry/CreateNewOrderEntry?... will only be matched by the last entry ("Default"), and given that both the controller and the action name are specified in the URL, this means that it will route directly to a CreateNewOrderEntry method on the OrderEntryController, which is reproduced below:

public class OrderEntryController : CustomController {

#region Injected Services

public OrderRepository OrderRepository { set; protected get; }

#endregion

[HttpGet]

public ActionResult CreateNewOrderEntry(string productCode, string currency) {

OrderEntry oe = OrderRepository.CreateNewOrderEntry(productCode, currency);

return View("ObjectEdit", oe);

}

...

}

Note the following:

* The OrderEntryController inherits from the Naked Objects CustomController. This is a convenience, not a requirement.
* Services can be injected into a controller, just as they are into domain objects.
* CreateNewOrderEntry(string productCode, string currency) picks up the two parameters from the URL.
* The method returns the ObjectEdit view of the newly created OrderEntry object, which we will now look at ...

[NotPersisted]

public class OrderEntry {

[Disabled, MemberOrder(10)]

public Product Product { get; set; }

...  
}

We can see that the OrderEntry object is a regular domain class, but has been marked up with [NotPersisted], indicating that this object will only ever be transient. (See the warning about Code First). Note that the user will still be presented with a Save button - but in this case it just takes the entered values into the object in memory, not the persisted store (database).

This is what is known in MVC terminology as a 'view model' - a domain object that exists solely for user interface purposes - it is not persisted.

This will be first presented to the user with the Views > OrderEntry > ObjectEdit view, shown here:

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.WithoutServices.Master" Inherits="System.Web.Mvc.ViewPage<LicenseManagement.OrderEntry>" %>

<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>

<%@ Import Namespace="LicenseManagement"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

<%: Html.ObjectTitle(Model)%>

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<div class="ObjectEdit" id="<%: Html.ObjectTypeAsCssId(Model) %>">

<%:Html.ValidationSummary(true, "Edit was unsuccessful. Please correct the errors and try again.")%>

<%:Html.UserMessages() %>

<h2>Enter Order Details:</h2>

<%

using (Html.BeginForm("Edit",

Html.TypeName(Model).ToString(),

new {id = Html.GetObjectId(Model).ToString()},

FormMethod.Post,

new {@class = "Edit"}))%>

<%:Html.PropertyListEditWithout(Model,"SubTotal", "VAT", "Total")%>

</div>

</asp:Content>

This is effectively just a standard object edit view, but with the Html.PropertyListEditWithout specifying that a number of properties are not to be shown at this stage (as they are calculated later).

Until you are familiar with the standard URL patterns generated by Naked Objects MVC, the best thing is to develop and run the application step by step, and view the HTML source in the browser. If we view the HTML source generated by the above view we can see that all the fields are contained within a form, which begins:

<form action="/OrderEntry/Edit?id=LicenseManagement.OrderEntry ... >

This tells us that the URL generated when the user hits Save, will be ...OrderEntry/Edit?... . This is picked up by this entry in RegisterRoutes:

routes.MapRoute("Saving details",

"OrderEntry/Edit",

new {controller = "Generic", action = "Edit"} );

This directs the flow to the Edit action on the Naked Objects GenericController which we know from general operation will bring up the standard ObjectView - with the values taken into the object. If we provide our own view in (Views > OrderEntry > ObjectView) this will be picked up automatically:

<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.WithoutServices.Master" Inherits="System.Web.Mvc.ViewPage<LicenseManagement.OrderEntry>" %>

<%@ Import Namespace="NakedObjects.Web.Mvc.Html"%>

<%@ Import Namespace="LicenseManagement"%>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">

<%: Html.ObjectTitle(Model)%>

</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

<div class="ObjectView" id="<%: Html.ObjectTypeAsCssId(Model) %>">

<%: Html.UserMessages() %>

<h2>Confirm Order Details:</h2>

<% using (Html.BeginForm("EditObject",

Html.TypeName(Model).ToString(),

new {id = Html.GetObjectId(Model).ToString()},

FormMethod.Post,

new {@class = "Edit"})) {%>

<%: Html.PropertyList(Model)%>

<%: Html.PropertyListEditHidden(Model, true)%>

<%}%>

<%-- A separate form for the other action, containing all props but hidden--%>

<% using (Html.BeginForm("ConfirmOrder", "OrderEntry",

new {id = Html.GetObjectId(Model).ToString()},

FormMethod.Post,

new {@class = "Edit"})) {%>

<%: Html.PropertyListEditHidden(Model, false)%>

<button type="submit">Confirm & Proceed to Payment</button>

<%}%>

</div>

</asp:Content>

This view contains two parts. The first is a form that provides the user with the Edit button (to go back and make changes). All the visible fields are displayed by <%: Html.PropertyList(Model)%>. However, because this is a not persisted object, we need to ensure that all the fields, including any hidden fields, are contained in the view (albeit some of them invisibly) - otherwise they will be lost. This is done with <%: Html.PropertyListEditHidden(Model, true)%> and it is the true parameter here that causes the Edit button to be rendered.

The second half of the view provides the user with the Confirm & Proceed to Payment button. This must be implemented as a separate form, and again, inside that form we must include all the properties of the not persisted model object using PropertyListEditHidden, but this time with the withEditButton parameter set to false.

If the user selects this button, then (from the "Default" routing above) be directed to the ConfirmOrder action on the OrderEntry controller, which is shown below:

[HttpPost]

public ActionResult ConfirmOrder(FormCollection fc) {

var oe = RecreateTransient<OrderEntry>(fc);

var order = oe.Confirm();

return Redirect(PaypalConstants.PaypalAddress + "?" + order.GeneratePayPalMessagePayload());

}

Because the OrderEntry object that we had created in the previous controller method was [NotPersisted] the session will have not record of it. We must therefore create a brand new instance of OrderEntry and copy all the data from the posted form into this new instance. RecreateTransient<OrderEntry>(fc) is just a shortcut, that saves us from having to write all that code. The method then goes on to call a Confirm method on that OrderEntry, the code for which is shown below:

public Order Confirm() {

//Set up Customer

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

cust.Name = Name;

// etc.

//Set up Order

var order = Container.NewTransientInstance<Order>();

order.Customer = cust;

// etc.

Container.Persist(ref order);

//Set up PayPal request

OrderService.OrderViaPayPal(order);

return order;

}

This method is copying data out of the not-persisted OrderEntry and into separate new Customer and Order objects. Container.Persist(ref order) will persist both the Order and the associated Customer object at the same time. Going back to the controller method above, we can see that it goes on to call a method GeneratePayPalMessagePayload on the (persisted) Order, and then uses this to re-direct the browser to the PayPal site to take payment.

## MVC - Additional How-Tos

### How to inject Services into Controllers and Views

Any service that is registered in the Run class may be injected into a Controller and/or into a View.

#### Injecting into Controllers

Services are injected into Controllers in exactly the same way as services are injected into domain objects - use the injs snippet.

#### Injecting into Views

To make services available to your views via injection, first create a sub-class of System.Web.Mvc.ViewPage, and add the properties for the injected services, again using the injs snippet. An example is shown below:

using System.Web.Mvc;

using AdventureWorksModel;

namespace RunMVCGeneric.Views {

public class ViewWithServices<T>: ViewPage<T>

{

public CustomerRepository CustomerRepository { set; protected get; }

public SalesRepository SalesRepository { set; protected get; }

}

}

Then ensure that each of your custom views that needs direct access to injected services, inherits from this new view class.

### How to encrypt hidden fields in a transient object

If you display a transient (not yet persisted) object that has some hidden properties (this is not a common scenario!) then those hidden properties will exist within the HTML, so that they can be posted back with the properties that the user has entered. (This is not the case for Hidden properties on persisted objects).

If this is a security concern (in other words if those hidden properties contain sensitive data) then it is possible to specify that any hidden properties within the HTML should be encrypted, using a pluggable encryption service. Naked Objects provides a default implementation: NakedObjects.Web.Mvc.Helpers.SimpleEncryptDecrypt. Alternatively, if you wish to use a specific form of encryption, you may create your own implementation of NakedObjects.Web.Mvc.IEncryptDecrypt. The service simply needs to be registered in the SystemsServices property on the Run class, as shown below:

protected override IServicesInstaller SystemServices {

get {

return new ServicesInstaller(new SimpleEncryptDecrypt());

}

}

### How to disable server-side validation (Ajax)

See [Executed](#_Executed)

## 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 the NakedObjects.Snapshot.XMLSnapshotService in the SystemServices property of your Run class:

protected override IServicesInstaller SystemServices {

get { return new ServicesInstaller(new 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).

# Creating and using a Restful Objects API

*This section of the manual is designed to be read in conjunction with the Restful Objects specification, which may be downloaded from restfulobjects.org.*

Restful Objects for .NET is an implementation of the Restful Objects specification for the Microsoft .NET platform.

It allows you to write a domain object mode in any .NET language and then expose the complete functionality of the domain model via a set of 'restful' resources, conforming to the Restful Objects standard.

Restful Objects for .NET is built on top of the Naked Objects for .NET framework, and therefore follows the same, very simple, programming conventions for domain object models. These conventions are described, in tutorial style, in Section 2.3, “Adding behaviour to your domain objects - a how-to guide”, and more formally in Section 2.4, “The Naked Objects programming model”.

## Adding a Restful Objects API to a Naked Objects MVC application

As of version 5.7 of the Naked Objects Framework, any MVC project that has the NakedObjects.Mvc-FileTemplatesNuget package installed into it, will have the ability to provide a Restful Objects API alongside the MVC user interface. For security reasons the Restful Objects API is not enabled by default. To enable it you just need to modify the RestRoot property on the RunWeb class, for example:

public static string RestRoot {

get { return "myrestfulapi"; }

}

The Restful Objects API will then be available on the specified URL segment. For example (of you are prototyping), the home page of the MVC application might be:

http://localhost:50326/System

and the home resource for the Restful API on

http://localhost:50326/myrestfulapi

If you wish to have the Restful API on a different port altogether, then you can do this by adding a standalone Restful Objects API as described in the next section.

## Creating a standalone Restful Objects API

If you do not want to have a Naked Objects MVC user interface for your application - just a Restful Objects API - then you may proceed as follows:

1. Create a new ASP.NET Web Application, specifying MVC as the template, and set this as the Startup Project for your solution. (We'll refer to this as the 'Run' project)
2. Find and install the NuGet package RestfulObjects.Server into this project
3. Add project reference(s) from the Run project to your Model project(s).
4. Open the RunWeb class (within the App\_Start folder). Within this Run class, register your services, and make any other changes you may require to the default run configuration.
5. Configure the Persistor property on the Run class as explained in: Section 2.1.1, “Creating a Model project from scratch”. If you are working with an existing database add the connection string(s) into the Web.config in your Run project.
6. Run the server project as a local app. This will launch a web-browser pointing at the 'home' resource for the application, running within LocalHost.

The home resource, as with all Restful Objects resources, will return a JSON representation. Microsoft Internet Explorer does not recognise JSON and so will offer to Save the returned representation as a file. We therefore recommend the use of the Chrome or Firefox browsers, with a suitable plug-in such as JSONView and/or REST Client.

## Conformance to the Restful Objects specification

The current release of Restful Objects for .NET, implements the full mandatory requirements of the Restful Objects specification version 1.1.0. The optional capabilities defined in that version of the spec, are implemented as follows (this information is available via the /version resource in the API):

* **blobsClobs** - supported..
* **domainModel** - Both the simple and the formal scheme are implemented, including the ‘selectable’ option. See [How to specify the scheme for domain model type information](#domain_model_type).
* **protoPersistentObjects** - supported. See [How to work with proto-persistent objects](#_How_to_work).
* **validateOnly** – supported
* **inlinedMemberRepresentations** - supported.
* **deleteObjects** - Not supported. (This is a deliberate policy. If you want to delete objects, simply provide an action method for deleting, either on an object or a service).

#### Lists and Sets

Although the Naked Objects programming model will recognise both Lists and Sets as multiple associations (i.e anything that implements IEnumerable<T>), Restful Objects for .NET treats all collections as being Lists. In other words, adding an object to a collection via the Restful Objects API will always require a POST method, even if the underlying type is in fact a Set. Note that this is the safe option. If you attempt to add the same object twice to a Set, using a POST, the second attempt will effectively be ignored. In a future release we expect to be able to allow objects to be added to a Set using PUT, as per the Restful Objects specification.

### Custom extensions

The Restful Objects specification permits the addition of custom extensions in defined places. Restful Objects for .NET adds the following custom extensions, where appropriate (all have the x-ro-nof prefix):

* x-ro-nof-serviceType- takes one of the values of menu, contributed, or system depending on how the service was registered in the Run class.
* x-ro-nof-renderInEditMode - true or false
* x-ro-nof-presentationHint - see [PresentationHint](#_PresentationHint) attribute
* x-ro-nof-mask- provides the string value specified using the Mask attribute.
* x-ro-nof-choices - provides a map of name values for a enum property or action parameter

## Restful Objects Server - additional how-to's

### 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

Restful Objects for .NET 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 specify the scheme for domain model type information

The Restful Objects specification defines two approaches to the supply of domain type information - the 'simple scheme' and the 'formal scheme'. Implementations may support none, either or both of these, with the option for the client to select between them. Restful Objects for .NET supports both schemes and unless the client request specifies which is required, both forms of domain type metadata will be included in every response. This provides maximum flexibility, but also increases the size of the returned representations; it also makes them harder for a human to read when debugging for example. A further issue is that the formal scheme may reveal information about the domain model that is considered sensitive. For example while the visibility of the Customer.EstmatedNetWorth property may be restricted using authorization, the fact that such a property exists within the system is visible via the domain type representation under the formal scheme. By contrast, under the simple scheme, if the property was hidden from a user, then the existence of such a property would also be fully hidden.

For all these reasons, Restful Objects for .NET allows the developer to restrict the options available on a particular deployment. This is done by adding the following line of code into the RegisterRoutes method on the NakedObjectsStart class in your run project:

RestfulObjectsController.DomainModel = RestControlFlags.DomainModelType.None;

where None may be replaced by any of the values defined in the DomainModelType enumeration: None, Simple, Formal, Selectable.

### 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 attribute as described in Section 2.3.6.1, “How to handle concurrency checking”.

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.Helpers.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), Restful Objects for .NET will return an http '301 Re-direct' to an alternative Restful Objects object resource that may even be on another server. That url for that alternative object resource is constructed from the values of the ServerName and Oid properties that the implementation of IRedirectedObject requires, as follows:

http://{ServerName}/object/{Oid}

This pattern is particularly effective as a mechanism for integrating multiple existing systems. In the following example, the Order object may be though of as existing in a 'new' system and the Customer in on another ('old') system. The Customer class on the new system 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 refer to a specific instance of a Customer object on the old system. The (optional) 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. (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 {

[Hidden, Key]

public virtual int Id { get; set; }

[Title]

public virtual string Title { get; set; }

#region Implementation of IRedirect

public string ServerName { get; set; }

public string Oid { get; set; }

#endregion

}

### How to change cache settings

Based on the three 'shorthand' types of responded 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. This will be added in a future release.

For the time being, if you need to use a Complex Type (perhaps because the same domain model is used by another application running Naked Objects MVC, for example) 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

k9nnb50069feds09llu5/66vb3ln99f3sx6kkp77hy

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 in the SystemsServices property of your Run class.

### 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:

Mark up certain classes, properties, or methods with a Hidden attribute. This is best used when you only wish to exclude small parts of the model from the Restful API. However, this approach will also hide the same parts of the model from any Naked Objects MVC user interface - which might not be your intent. In which case consider the other patterns ...

* 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 Naked Objects MVC 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. 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.GetTypeFromLoadedAssemblies(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.GetTypeFromLoadedAssemblies 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 RegisterRoutes method on the RestfulObjectsStart class:

RestfulObjectsControllerBase.AcceptHeaderStrict = false;

With strict Accept header enforcement switched off, the generated API is no longer strictly conforming to the Restful Objects spec. 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);

}

### How to work with addressable View Models

Section 2.2 of the Restful Objects specification (v 1.0) introduces the idea of addressable View Models, and section 32 provides a code sketch as to how this might be supported. The Restful Objects Server (as of v 1.1) does support this pattern using an approach broadly similar to that code sketch.

See View Model.

### 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.

# Running without a user interface

It is possible to run Naked Objects without any user interface, for example to implement batch operations, or a custom publish & subscribe mechanism.

## Running 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:

internal class Program {

private static void Main(string[] args) {

RunExe.Run();

}

}

1. In the RunExe (extending from RunBatch) class register your domain services, and configure other options such as the Persistor - just as you would do for any Run class in Naked Objects. Note that the Run method contains the following:

public static void Run() {

new RunExe().Start(new BatchStartPoint());

}

1. The package will 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...

## Running multiple threads asynchronously

.NET 4 introduced a new async capability. Naked Objects now leverages this to provide a convenient mechanism for running multiple Naked Objects threads in parallel. To use this capability, 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 interface (as it does for IDomainObjectContainer) NakedObjects.Async.AsyncService. You will need to register this (or another implementation if you wish to write your own) in the SystemsServices property of your Run class.

Within your code, 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 void RunAllProcessesDueBy(DateTime dateTime) {

IList<BatchProcessDefinition> due = BatchRepository.ListProcessesWithNextRunsDueBy(dateTime);

var tasks = due.Select(bpd => AsyncService.RunAsync(() => BatchRepository.FindAndRunProcessDefinition(bpd.Id))).ToArray();

Task.WaitAll(tasks);

}

In the example above, the BatchRepository and BatchProcessDefinition are straightforward domain classes. The first line just returns a set of objects that contain details of tasks due to be run by the specified DateTime. The second line invokes a method on BatchRepository 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 not attempt to 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, FindAndRunProcessDefinition is passed in the Id of the BusinessProcessDefinition, not the domain object itself.

**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 (and therefore within an MVC run project).

# 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.
* **WIF Claims-based Authorization**. Naked Objects provides an implementation of the WIF-defined Claims Authorization Manager for this purpose. WIF Claims-based Authorization may only be used with WIF-managed authentication.

## 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

The following example shows how you can add your-own custom logic to manage authorization. Start by overriding the Authorizer property on the Run class:

protected override IAuthorizerInstaller Authorizer {

get {

return new CustomAuthorizerInstaller(new MyDefaultAuthorizer(), new FooAuthorizer(), new BarAuthorizer());

}

}

The constructor for the CustomAuthorizerInstaller has the following signature:

public CustomAuthorizerInstaller(ITypeAuthorizer<object> defaultAuthorizer, params object[] typeAuthorizers)

The defaultAuthorizer parameter is mandatory, followed by zero or more typeAuthorizers. Each of the typeAuthorizers should implement 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).

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 void Init() {...}

public bool IsEditable(IPrincipal principal, Foo target, string memberName) {...}

public bool IsVisible(IPrincipal principal, Foo target, string memberName) {...}

public void Shutdown() {...}

}

* Init and ShutDown are hook methods to allow you to insert specific logic when the framework is initialised and shut-down respectively. The implementation of these two methods may be left empty if you have no need for your own specific initialisation logic.
* 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.
* IsEditsable 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.

Any implementation of ITypeAuthorizer<T> or INamespaceAuthorizer, that is instantiated via the CustomAuthorizerInstaller (as shown above) 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.

## WIF Claims-based authorization

Naked Objects MVC provides an example implementation of the Claims Authorization Manager that is defined by WIF - it is likely that you will want to write your own, but you can use the source code of our example to get started. To specify that authorization should be delegated to this example implementation, add the following to the RunWeb class (this will require that your project has a reference to NakedObjects.Authorisation.Wif.dll):

protected override IAuthorizerInstaller Authorizer {

get {

return new WifAuthorisationInstaller();

}

}

The Claims Authorization Manager expects the claims to be defined within the Web.config file, as shown below:

<configuration>

...

<microsoft.identityModel>

<service>

<audienceUris>...</audienceUris>

<federatedAuthentication>...</federatedAuthentication>

<applicationService>

<claimTypeRequired>

<!--Claim types defined here - generated by the STS-- e.g.:>

<claimType type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role" optional="true" />

</claimTypeRequired>

</applicationService>

<issuerNameRegistry ...>...</issuerNameRegistry>

<certificateValidation certificateValidationMode="None" />

<claimsAuthorizationManager ...>

<!--Claims defined here-->

</claimsAuthorizationManager>

</service>

</microsoft.identityModel>

...

</configuration>

Most of this should have been set up when you ran the Add STS Reference... wizard, to set up WIF-managed authentication, including the definition of the claim types (such as the role type shown above) that are recognised by the STS. The XML that needs to be added in is the claimsAuthorizationManager element. An excerpt from an example implementation is shown below:

<claimsAuthorizationManager

type="NakedObjects.Reflector.Security.Wif.NakedObjectsClaimsAuthorisationManager,

NakedObjects.Authorisation.Wif, Version=1.0.0.0, Culture=neutral">

<class name="Address" fullname="AdventureWorksModel.Address">

<member name="AddressLine1" type="ViewField">

<claim value="Customer Service" type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role" />

<claim value="Manager" type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role" />

</member>

<member name="AddressLine1" type="EditField">

<claim value="Manager" type="http://schemas.microsoft.com/ws/2008/06/identity/claims/role" />

</member>

<member ...

...

</class>

<class ...

...

</claimsAuthorizationManager>

Note the following:

* NakedObjectsClaimsAuthorisationManager is specified as the implementation of the (required) claimsAuthorizationManager.
* A class element defines a domain class (either an entity type or a service), defined by its fully-qualified name.
* The class element contains a number of member elements that define the public members (properties or methods) of that class. Each member is defined by its name and type, of which there are three: ViewField, EditField (each property is represented twice: once to define the viewing claims and separately for the editing claims) and Action.
* Each member element may contain one or more claim elements. The claim element is defined by the claim type (which must correspond to one of the claim types recognised by the STS (as specified in the claimTypeRequired element above), and by a value. The list of claims in each member is an OR list - the user must satisfy at least one of those claims in order to be authorized for that member. If the member element is empty (contains no claims) then any user is deemed to be authorized. However, if any member (or a whole class) in the model does not have a corresponding element in the .config then that member is deemed to be unavailable to any user - this is a safeguard policy.

The example snippet above specifies that on the AdventureWorksModel.Address class, the AddressLine1 property may be viewed by any user that has the role Customer Service or the role Manager, but that that same property may only be edited by a user with the role Manager.

In order to be able to edit a property, the user must also be authorized to view that property - hence in the above example, the Manager claim appears in both the view and edit definitions for AddressLine1.

# 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 install one or more audit services via the Auditor property of the Run class, as illustrated in the following example:

protected override IAuditorInstaller Auditor {

get {

return new CustomAuditorInstaller(new MyDefaultAuditor(),  
 new SalesAuditor(), new ProductsAuditor());

}

}

The constructor of the CustomAuditorInstaller requires that the first parameter (MyDefaultAuditor above) must implement NakedObjects.Audit.IDefaultAuditor; followed by an unlimited number of implementations of NakedObjects.Audit,INamespaceAuditor, each specifying a particular namespace of interest. 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 updatedObject);

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 services implementing IAuditor may manipulate the provided details of the user action and then optionally persist them as domain objects. Note that any implementation of IAuditor may have the domain object container or other services injected into it for this purpose. The persisted audit records may then be made available to suitably-authorised users via, say, an AuditRepository service.

Any implementation of INamespaceAuthorizer, that is instantiated via the CustomAuditorInstaller (as shown above) can have domain services and/or the IDomainObjectContainer injected into it.

# 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 Visual Studio test framework.

Create an XAT project as follows:

1. Create a new MS Test project.
2. Invoke Manage Nuget Packages, find and install the NakedObjects.Xat package into this project

### Creating an XAT test class

As well as references to the necessary Naked Objects assemblies, the project will now also contain an ExampleXAT class. The top of this class looks like this:

[TestClass()]

public class ExampleXAT : AcceptanceTestCase {

[TestInitialize()]

public void Initalize() {

InitalizeNakedObjectsFramework();

}

[TestCleanup()]

public void Cleanup() {

CleanupNakedObjectsFramework();

}

protected override IServicesInstaller Services {

get {

return new ServicesInstaller(

new EmployeeRepository(),

new ClaimRepository(),

new RecordedActionRepository(),

new RecordActionService(),

new RecordedActionContributedActions(),

new DummyMailSender());

}

}

protected override IFixturesInstaller Fixtures {

get {

return new FixturesInstaller(new RefdataFixture(), new EmployeeFixture(), new SvenClaims\_All());

}

}

We can see that the class inherits from AcceptanceTestCase (which forms part of the Naked Objects XAT framework), and that it is marked up with a TestClass attribute. Each of the test methods (e.g. FindMyClaims() is marked up with a TestMethod attribute. There are also two methods (Initalize() and Cleanup() ) that have been marked up with the TestInitialise and TestCleanup attributes respectively - these are automatically run before/after each test. You can add your own code into these methods to initialize and cleanup common variables between tests. It is important that these methods call the InitializeNakedObjectsFramework() and CleanupNakedObjectsFramework() respectively, both of which are provided on AcceptanceTestCase.

In addition, there are overridden properties that specify any services (such as repositories) and/or data fixtures that you want to be available for the tests - this is equivalent to specifying services and or data fixtures in the Run class of your application - and it may be convenient to copy the relevant code from that Run class. You may also override any other properties from the Run class, for example the Peristor, the Authorizer and so on.

You can create a new class that follows this outline using the XAT Item template.

### Writing tests

The following code shows the example test method from within the ExampleXAT class (some lines have been omitted for space). 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[] tp1 = claim.GetAction("Copy All Expense Items From Another Claim").Parameters;

tp1[0].AssertIsMandatory().AssertIsNamed("Claim or Template").AssertIsDescribedAs("");

tp1[1].AssertIsOptional().AssertIsNamed("New date to apply to all items").AssertIsDescribedAs("");

ITestParameter[] tp2 = claim.GetAction("Create New Claim From This").Parameters;

tp2[0].AssertIsMandatory().AssertIsNamed("Description").AssertIsDescribedAs("");

tp2[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);

. . .

claim.GetContributedAction("All Recorded Actions", "Recorded Actions").AssertIsValidWithParms();

}

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.

### 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

As an alternative to running your tests in-memory with fixtures, you may also elect to run them against a database. Follow these steps:

1. Within your test class, specify that you want to run with the Entity Persistor, in the same way that you would specify this in a Run class (by overriding the Persistor property.)
2. Ensure that you have specified the connection string to the database in an App.config file within the XAT project. (Typically, you will copy this from the App.config generated by your domain projects.

#### 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.

## End to End Testing with Selenium

When using Naked Object MVC, user-interface testing is not always necessary, or even valuable:

* If you are using only generic views you can rely on the fact that Naked Objects MVC 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 are using any custom views or controllers 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.

Selenium is browser-based automated UI test framework. Naked Objects provides a set of helper methods that can facilitate the writing of Selenium tests specifically to work with a Naked Objects MVC application.

### Adding a Selenium test project

1. Add a new Test Project to your solution using the standard Microsoft Test Project template. There is no need to add any references to any other projects in the solution, because you will be testing against an application running within a local (or remote) web server.
2. Find and install the NuGet package NakedObjects.Mvc.Selenium into your test project; this will automatically install the Selenium framework and other dependencies.
3. In order to run your Selenium tests, you will first need to run the application on local host, or deploy it to a remote IIS. If running in local host, take a note of the server url including the port number e.g. http://localhost:53686/
4. On the Site.WithServices.Master view, comment out the references to any .css files. The Selenium tests ignore .css, but you will find that the drop-down menu implementation on the default .css will render the tests inoperable because it depends on mouse-location to un-hide menu items.
5. Then simply run the tests as you would run regular unit tests.

### Writing Selenium tests

The following shows an example of a Selenium test that is written to run against Naked Objects MVC:

using NakedObjects.Web.UnitTests.Selenium;

using OpenQA.Selenium;

using OpenQA.Selenium.IE;

[TestClass]

public class MyTests {

protected const string url = "http://localhost:53686/"; //Replace the url with your application server url

protected IWebDriver br;

[TestInitialize]

public virtual void InitializeTest() {

br = new InternetExplorerDriver();

br.Navigate().GoToUrl(url);

}

[TestMethod]

public void FindEmployeeByName() {

br.ClickAction("EmployeeRepository-FindEmployeeByName");

br.GetField("EmployeeRepository-FindEmployeeByName-Name").TypeText("paul", br);

br.ClickOk();

br.AssertContainsObjectView();

br.AssertPageTitleEquals("Paul, 190");

}

}

Some points to note:

* This test is written to use Internet Explorer. Selenium drivers for the Firefox and Chrome browsers are also available.
* The class NakedObjects.Web.UnitTests.Selenium.SeHelpers defines a large number of extension methods for IWebDriver and IWebElement, a few of which are shown in the example above (e.g. ClickAction, ClickOK). These mimic the gestures typically applied to a Naked Objects MVC user interface.
* The identifiers provided as arguments to several of these helper methods are the Ids found in the generated HTML

#### Writing tests to run on multiple browsers

You can write your tests to run against multiple browsers, using the following pattern:

public abstract class MyAbstractTests {

protected const string url = "http://localhost:53686/"; //Replace the url with your application server url

protected IWebDriver br;

protected void CommonInitialize() {

br.Navigate().GoToUrl(url);

}

public abstract FindEmployeeByName();

protected void CommonFindEmployeeByName() {

br.ClickAction("EmployeeRepository-FindEmployeeByName");

br.GetField("EmployeeRepository-FindEmployeeByName-Name").TypeText("paul", br);

br.ClickOk();

br.AssertContainsObjectView();

br.AssertPageTitleEquals("Paul, 190");

}

}

[TestClass]

public class MyIETests {

[TestInitialize]

public virtual void InitializeTest() {

br = new InternetExplorerDriver();

CommonInitialize();

}

[TestMethod]

public override void FindEmployeeByName() {

CommonFindEmployeeByName();

}

}

[TestClass]

public class MyFirefoxTests {

[TestInitialize]

public overrides void InitializeTest() {

br = new FirefoxDriver();

CommonInitialize();

}

[TestMethod]

public override void FindEmployeeByName() {

CommonFindEmployeeByName();

}

}

# Internationalisation / Localisation

Naked Objects has full support for internationalisation (sometimes referred to as 'i18n') and localisation ('l10n') - allowing an application to be presented in different languages and/or cultures for different users without having to alter the domain code.

Following standard .NET practice, localisation is specified in a set of .resx files, which are compiled into their own assemblies. If you look in the bin directory of your Run projects you will see three Resources .dlls as follows:

* NakedObjects.PMResources.dll (the resources associated with the NakedObjects.ProgrammingModel NuGet package)
* NakedObjects.MVCResources.dll (the resources associated with the NakedObjects.Mvc NuGet package)
* NakedObjects.FWResources.dll (the resources associated with the NakedObjects.Framework NuGet package, on which the Mvc package depends)

## Adding the Resources projects to your solution

The first step toward localisation is to locate the source code projects that create these three assemblies (download them from CodePlex):

* NakedObjects.PMResources (the resources associated with the NakedObjects.ProgrammingModel NuGet package)
* NakedObjects.FWResources (the resources associated with the NakedObjects.Framework NuGet package)
* NakedObjects.MVCResources (the resources associated with the NakedObjects.Mvc NuGet package)

and to add these three projects to your solution.

Opening each project will show the .resx files as shown here:



The role of these is as follows:

* Model.resx contains the various labels that are generated reflectively from your own model project(s) - object names, property names, action names, parameter names and so on. The version of this file included in the NakedObjects.Resources project is empty by default. You will need to generate the file from your application - which is a very simple process, described below.
* NakedObjects.resx specifies all the strings used in the core Naked Objects framework. Typically these are generic validation, warning or error messages such as 'Field not editable'.
* MvcUI.resx specifies all the strings used within the generic Naked Objects MVC user interface, such as the labels for the Edit, Save and Select buttons, and the generic actions in the Find menu.
* ProgrammingModel.resx specifies all the strings used in the (optional) Helper classes.

In addition to these standard resource files, you may wish to add further resource files, for example to cover strings used within your model code (such as application-specific error, warning or validation messages), or for text used in any custom views. Any such additional resource files will need to be referenced using the standard .NET patterns for localisation.

## Specifying that you wish to localise the application

By default a Naked Objects application is not localised. However, date and currency formats will always be localised to the formats specified on the user's machine.

To localise the application set the Localize property on the Run class and set it to return true:

Protected Overrides ReadOnly Property Localize() As Boolean

Get

Return True

End Get

End Property

protected override bool Localize {

get {

return true;

}

}

Then add the following method into the Global.asax.cs file, to tell the system to pick up the language/locale from the browser's settings, and if none is specified by the browser to default to English for the US locale (or whatever your preference):

protected void Application\_BeginRequest(object sender, EventArgs e) {

using (var fakePage = new Page()) {

var ignored = fakePage.Server;

fakePage.Culture = "auto:en-US";

fakePage.UICulture = "auto:en-US";

}

}

## Populating the Model resource file

On your Run class, override the ModelResourceFile property and return the full-qualified path name for the Model.resx file in the NakedObjects.FWResources project, for example:

Protected Overrides ReadOnly Property ModelResourceFile() As String

Get

Return "C:\MyApp\NakedObjects.FWResources\Model.resx"

End Get

End Property

protected override string ModelResourceFile {

get {

return @"C:\MyApp\NakedObjects.FWResources\Model.resx";

}

}

Now run the application. The file will be created during the initial start-up phase of the application, but the file will only be terminated when the application is properly terminated. When running Naked Objects MVC you should stop the Visual Studio Test Server and wait a few seconds to allow the server to finalise. The created file will over-write any existing Model.resx file.

This resource file will automatically include all items of all domain classes that can be reached by following code paths from any of the services that are registered via the Run class. Usually, this covers the entire domain model. However, it is possible, if you are using some sophisticated patterns that there are classes that can't be reached this way - in which case they will not be automatically added to the .resx file. You can always add items manually to the file afterwards. Another option is simply to ensure that during your translation run, you navigate to each one of the missing classes via the user interface at some point.

You should ensure that this overridden property is removed, or commented-out, for the live distribution. The safest policy is only to include the override when you are ready to create a Model.resx file that you wish to translate, and then exclude it again.

## Translating the resource files

The next step is to create a translated version of each of the .resx files for each of the locale that you wish to support. For example, to add support for the French (fr) language, you will need to end up with these files:

* Model.fr.resx
* NakedObjects.fr.resx
* MvcUi.fr.resx
* ProgrammingModel.fr.resx
* (Optionally) Your own resource file(s) such as MyAppStrings.fr.resx if you wish to localise strings within your own application code or custom views.

fr is the generic designation for the French language, whereas fr-FR is French specifically for the France locale, and fr-BE French language for Belgium locale. If you make your resource file specific to, say, fr-FR then it will not be picked up by browsers set to fr (though the reverse does work). This is all standard localization practice, not specific to Naked Objects.

You can copy and edit the .resx files by hand, or use a third-party translation tool or online service. The screenshot below shows a small part of a translated .resx file for the AdventureWorks sample model:



The translated resources files must be included in the projects, for example as shown below:



Finally, in your Run project, remove the direct references to the three resource .dlls, and replace these with project references to the three resources projects shown above.

When done, re-build the entire solution. If you show the bin directory of your Run project (hit Refresh if necessary) then you should find that it now contains:

* The original resources .dlls.
* A folder for each language for which you added translated .resx files

This is shown in the example below (other .dlls have been removed, for clarity):



## Testing your localised application

You are ready to test your localised application. (Remember to remove or comment out the ModelResourceFile property if you haven't already done so, but to leave the overridden Localise property set to true - see above). Set the language preference on your browser (how this is done varies between browsers). Then run the application as usual and check for the translated labels throughout.

# Troubleshooting

This section contains hints and tips on troubleshooting.

## Logging

Logging is managed through the Common.Logging framework. To add logging to your application you need to:

1. Add your choice of a common logging implementation into your Run project. Search the NuGet Public Gallery for 'common.logging', which will list a number of implementations including: Log4Net (in multiple versions), NLog, and Elmah.
2. Also in your Run project, find the class NakedObjectsStart, and the method InitialiseLogging().
3. Add code into this method appropriate to the implementation you have added. For example, if you have added one of the Log4Net implementations, you might add the following code:

var properties = new NameValueCollection();

properties["configType"] = "INLINE";

properties["configFile"] = @"C:\Naked Objects\nologfile.txt";

LogManager.Adapter = new Common.Logging.Log4Net.Log4NetLoggerFactoryAdapter(properties);

1. Again, depending on the implementation you added, you may wish/need to add some logging configuration into the .config file.

Having set up for logging, the Naked Objects Framework will log a significant number of generic internal events. You may also choose to add you own custom logging events from within your domain code.

## Problems running the AdventureWorks example application

### Error locating server

If you get this error message, it is likely to be because the wrong data source is specified in the connection string, which is located in the App.config file.

### The application runs very slowly

With the exception of the very first access to the database (which is always slow) the Naked Objects AdventureWorks application should run at a very good speed. The application will run more slowly if you run it in Debug mode from Visual Studio - especially so if you are connecting to a remote database server. It is strongly recommended that you run the application as an executable from Visual Studio (**Ctrl-F5**) unless you specifically need to debug the application.

## Problems working CodeFirst

### 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 services that are registered in the Run class. 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.

## 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 any services in the Run class.

### 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. Visual Studio creates Visual Basic classes public by default, but creates C# classes as internal by default.

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 (which is standard programming practice). See [Collection properties](#collection_property). The best way to avoid this error occurring is to create new collections using the coll code snippet.

### Could not find Entity Framework context for type

This is most likely to occur if you are working with multiple entity models (see Section 2.3.6.5, “How to work with multiple databases”). You need to ensure that:

* Each .edmx file has a different name (for example, based on the assembly name for the project within which it lives)
* Each generates a different connection string name.
* The connection strings are all copied into the App.config file within the run project.

## 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 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. Alternatively, just mark up the complementary method with the Hidden attribute.

## 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

The most likely explanation is that you haven't registered the service in your run-class. See also the point above ('injected Container is null'), which may apply to services also.

### The application runs OK in execute mode, but throws an exception in debug mode

Because Naked Objects makes heavy use of reflection, Visual Studio cannot always detect that an exception being thrown by one part of the framework is going to be picked up by another part of the framework, and will consequently break (when in debug mode) when the exception is thrown but before it is caught. Hitting Continue will typically work. If this behaviour proves annoying then you can change the settings on Visual Studio. Navigate to Debug > Exceptions > Common Language Runtime Exceptions and un-check the User-unhandled checkbox: