NAKED OBJECTS

Version 12

Application Developer Manual

Updated: 31st August 2021

Contents

What’s new in Version 12

Version 12 of the Naked Objects framework, offers the following new features of value to the application developer:

- Offers the option to work with either Entity Framework Core or Entity Framework 6. See Configuring the EntityObjectStore.
- Allows Properties or Collections to be contributed to an object, in a similar manner to ‘contributed actions’ using the new `[DisplayAsProperty]` attribute. See Contributed actions, properties or collections.
- Allows actions whose purpose is to edit one or more properties on a persistent object to be annotated with the new `[Edit]` attribute, and hence to allow (on the generic client) the action to be invoked using the ‘edit’ icon next to any of the fields, and to edit the property(ies) in situ rather than via a separate dialog. See Edit.

The structure of the framework has changed somewhat, though this should have no impact on domain code. In particular, the framework now separates `NakedObjects` components from `NakedFramework` components. Part of the motivation for this change is to permit the idea of running the framework with alternative programming models (of which ‘Naked Objects’ is just one), either for different applications or potentially even within a single application. Naked Objects Group is currently at at advanced stage of development on one such new programming model, using Functional Programming.

Getting started with the Naked Objects Template solution

We strongly recommend that you start from the Template solutions as explained below.

The Template Server solution

The Template Server solution for Naked Objects may be found here: https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Template/Naked%20Objects%20Server . You can either clone the repository, or just download and unzip the file: `Template.Server.zip`.

Open the `Template.Server.Sln` in Visual Studio (2019 or later).

The `Template.Server` solution contains two projects – `Template.Model` and `Template.Server`. This separation into two projects is just for clarity and flexibility, it is not a requirement.

The `Template.Model` project contains the domain object model, and provides a few classes as useful starting point:

`Student, Teacher, Set, Subject, SubjectReport` – domain entity types that will be persisted to the database using EntityFrameworkCore.

`StudentRepository, TeacherRepository, SetRepository, SubjectRepository` stateless service class that provides methods to create and retrieve entities.

`ExampleDbContext` – follows the standard patterns for a DbContext designed to work with EntityFrameworkCore, including (optionally) the ability to create seed data.

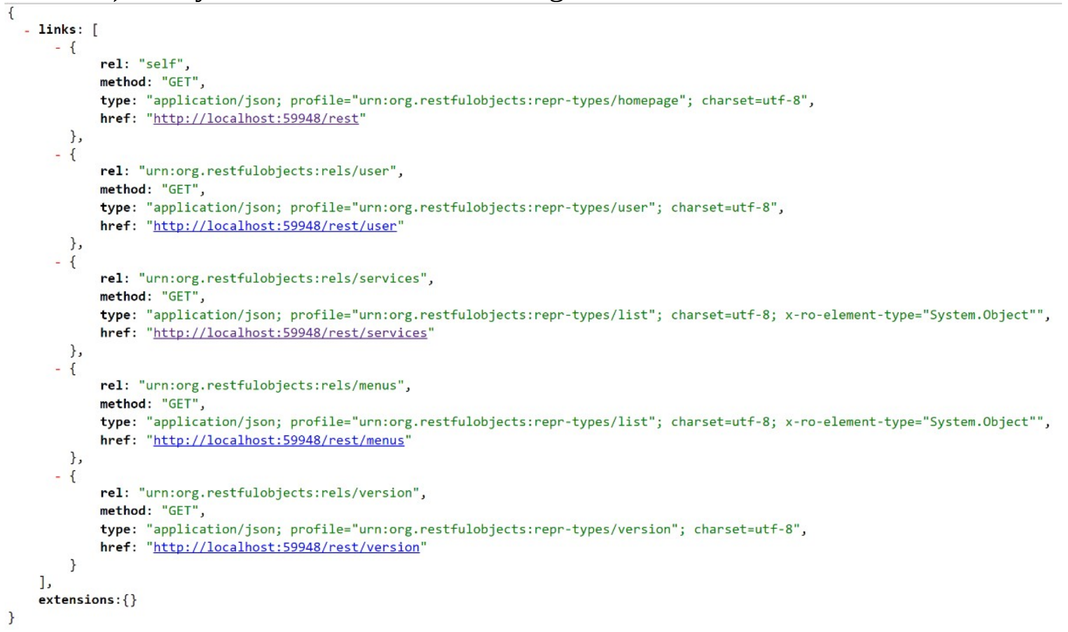
`ModelConfig` – a static class that is a simple mechanism by which the framework can obtain an array of all the domain entity `Types`, the `Services`, the `MainMenu` definitions (which for simple prototypes *can* be the same as the services but for more complex systems will be separately defined) , and a function (`EFCoreDbContextCreator`) that can create an instance of the `DbContext` whenever needed. The (optional) use of a ModelConfig class like this means that changes to the Model project do not require *any* changes to the `Template.Server` project.

The `Template.Model` is not dependent upon the Naked Objects framework, only upon the `NakedObjects.ProgrammingModel`, which defines a simple API.

The `Template.Server` project is the start-up project, which when run creates a server using the Naked Objects framework. `Template.Server` depends upon `Template.Model`: at run time, the Naked Objects framework inspects the capabilities of the model and then makes calls into the domain code – not vice versa.

The `Template.Server` project contains generic code and system configuration. It will run with the `Template.Model` project without any code modifications, even as you modify or extend the model. Once you get to the stage of deploying an application then you will typically need to do some more system configuration in the server project, following standard ASP.NET Core patterns. See Application configuration.

Check that the Template.Server project is set as the start-up project, and then run it. After a short delay (the Template is having to create a database in the background first) your browser should open on this URL: `http://localhost:5000`. Note, again, that *this is the Url of the server, not the client*. It gives a view of the RESTful API to the Naked Objects server (you may download the full specification for this RESTful API from here). If your browser is configured to be able to display JSON (for example, Chrome with the JSONView extension installed) then you should see the following JSON.



```
{
  - links: [
      - {
            rel: "self",
            method: "GET",
            type: "application/json; profile="urn:org.restfulobjects:repr-types/homepage"; charset=utf-8",
            href: "http://localhost:59948/rest"
        },
      - {
            rel: "urn:org.restfulobjects:rels/user",
            method: "GET",
            type: "application/json; profile="urn:org.restfulobjects:repr-types/user"; charset=utf-8",
            href: "http://localhost:59948/rest/user"
        },
      - {
            rel: "urn:org.restfulobjects:rels/services",
            method: "GET",
            type: "application/json; profile="urn:org.restfulobjects:repr-types/list"; charset=utf-8; x-ro-element-type="System.Object"",
            href: "http://localhost:59948/rest/services"
        },
      - {
            rel: "urn:org.restfulobjects:rels/menus",
            method: "GET",
            type: "application/json; profile="urn:org.restfulobjects:repr-types/list"; charset=utf-8; x-ro-element-type="System.Object"",
            href: "http://localhost:59948/rest/menus"
        },
      - {
            rel: "urn:org.restfulobjects:rels/version",
            method: "GET",
            type: "application/json; profile="urn:org.restfulobjects:repr-types/version"; charset=utf-8",
            href: "http://localhost:59948/rest/version"
        }
    ],
    extensions: {}
}
```

If your browser is not configured to display JSON, you might get a message such as ‘Do you want to open or save rest.json?’ - which you should ignore. It is not necessary to be able to view the RESTful API directly, though it does provide useful visual feedback that the server has started up correctly.

The Template Client solution

The Template Client solution for Naked Objects may be found here:
https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Template/SPA

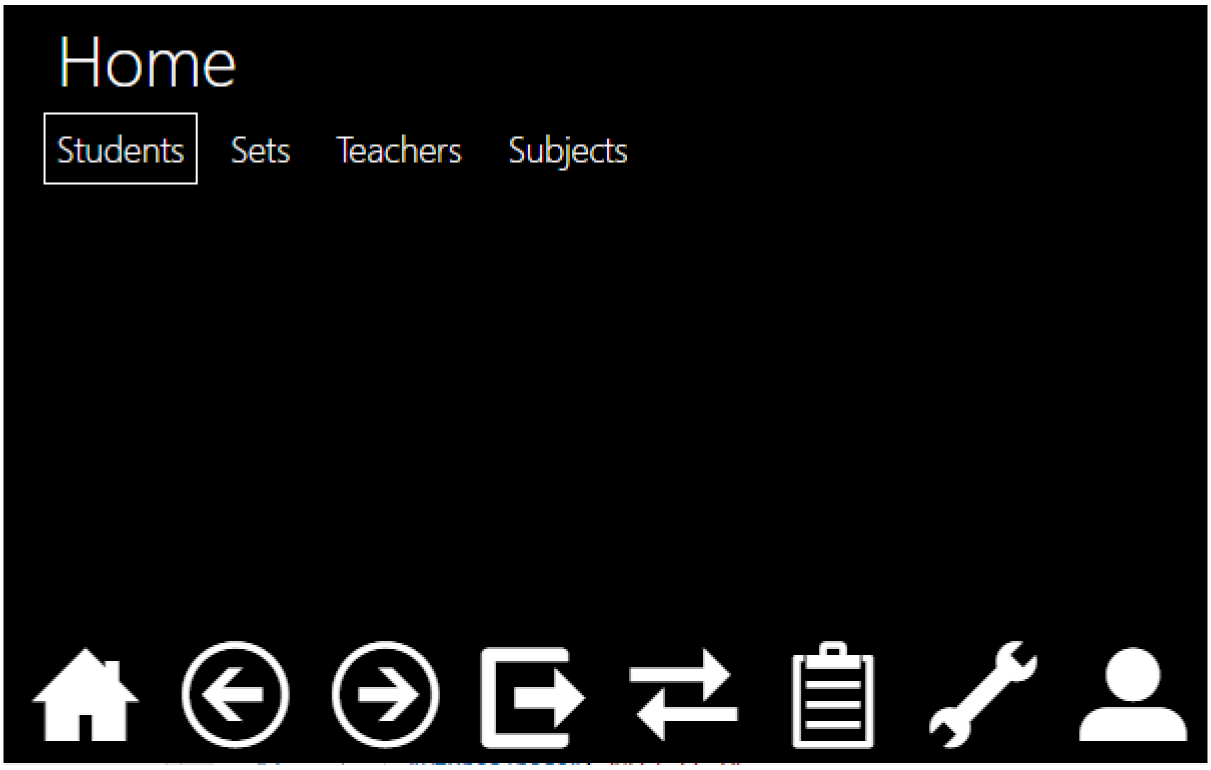
%20Client . You can either clone the repository, or just download and unzip the file: `Template.Client.zip`.

Open the `Template.Client.Sln` in a *separate instance of Visual Studio*. This is so that the two solutions may be run in parallel.

The `Template.Client` solution contains a single project of the same name. This is a standalone web project that provides a Single Page Application (Spa), which is pre-configured to connect to the the Server API mentioned above. (And the Server project is pre-configured, using CORS, to accept requests from the Url that the client deploys to). *To use the client you will need to have Node.js installed and up to date.*

Build and run the `Template.Client` project. Note that the first time you build might take several minites as the template project must first download and built multiple `NPM` packages. (Note: if you are already familiar with building and running Angular applications, you can perform the build process much faster using Visual Studio Code.)

When the build process is completed a browser should open on the URL `http://localhost:5001` . You should now be seeing the `Home` screen for the application. If you run the Client while the server is still starting up, you may see a 'Loading…' message at the bottom of the home screen, after which the main menus should appear:



This is a minimal application, with tiny functionality only. But (starting by clicking on the menu), try retrieving all the students, clicking on one of them, then creating a new student. The 'home' icon at the bottom of the screen takes you back to the home view, and you can experiment with what the other icons at the bottom of the screen do.

Next you should explore the relationship between the code and the user interface.

Troubleshooting

If the Template project does not fire up as expected, we suggest you investigate the following:

1. By default the `Template.Server` project is set up with logging on. Look for the file `nakedobjects_log.txt` within the project (you will need to Show All Files). Once running, errors should show up on an error view on the client, but errors that occur during the start-up process cannot typically be displayed on the client, so they are send to the log file. (Note that for security reasons, before deploying the server live, you should disable the passing of debugging information to the client. See Error: Reference source not found).
2. The most common source of start-up errors is database connection: a failure to connect to the database server, or to the specified database within the server. Check the connection string in the `appsettings.json` file of the server project. By default this is set up to work with `localDB`, which ships with Visual Studio, but you may prefer to change this to work with `SQLExpress` if you have it installed.
3. The next most common cause is a failure to create the `DbContext`. The `ExampleDbContext` in the `Template.Model` project is set up to drop and re-create the database every time the model changes. All it takes to prevent this from happening will be to have another live connection to that database - for example browsing it via the Server Explorer on Visual Studio.
4. Because the Server and Client are run as two separate projects, on two separate Urls (`localhost:5000` and `localhost:5001` respectively) it is necessary to have CORS (Cross-Origin Resource Sharing) set up. Specifically, the server must explicitly allow requests from the client Url – so if you are running the client on a different port, or different Url you will need to configure CORS. See Configuring cross-origin resource sharing (CORS).
5. Similarly, if you change the port or the Url for the server project, you will need to configure the client to point at the correct Url to find the RESTful API. This is done in the `config.json` file of the client thus:

```
{
    "appPath": "http://localhost:5000"
}
```

Updating the packages

You should periodically check for updates to the package.

The `Template.Server` project depends upon the `NakedObjects.Server` NuGet package, and you can check for updates via the NuGet Package Manager or the Package Manager Console.

The `Template.Client` project depends upon the following NPM packages:

@nakedobjects/gemini
@nakedobjects/cicero
@nakedobjects/view-models
@nakedobjects/services
@nakedobjects/restful-objects
@angular/cdk

These may be updated by opening a Command Prompt in the `Template.Client` project folder. `Npm – h` will give you a list of npm commands.

Writing a new application

We recommend that you create your domain model in separate project(s) as per the template. A domain model project needs to have the `NakedObjects.ProgrammingModel` NuGet package installed, but not any other part of the Naked Objects Framework. If you define your `DbContext` in the same project you will need to install the NuGet package for `EntityFrameworkCore` (or `EntityFramework 6` if you have configured the server project for that).

Now start to create your domain model, following the patterns and conventions described in Domain model - programming concepts and A how-to guide, and using the short-cuts described in See Using the Naked Objects IDE.

To turn your domain model into an application you will need to add a and server project into your solution, and add references to the domain model. By far the best way to so this is to start from a copy of the Template project.

You will also need to configure the application, following the guidelines in Application configuration .

The Naked Objects Client

Please note that this section has now been replaced by a separate manual titled: The Naked Objects Client: Configuration and Customisation. This may be downloaded from: https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Documentation

The Naked Objects Server

Application configuration

Application configuration follows the standard patterns of an ASP.NET Core Web API project, including the use of ASP.NET Core Dependency Injection to configure the services (components) of the system.

Use The Template Server solution as your starting point. In the `Startup.cs` class in you can see that the `ConfigureServices` method includes the following code to set up the NakedObjects framework and specify how it can find the **`DomainModelTypes`**, `DomainModelServices`, `MainMenus`, and the function to create the `DbContext`.

```
services.AddNakedFramework(frameworkOptions => {
    frameworkOptions.MainMenus = MenuHelper.GenerateMenus(ModelConfig.MainMenus());
    // builder.AddEF6Persistor(persistorOptions =>
                 { persistorOptions.ContextInstallers = new[] { ModelConfig. }; });
    frameworkOptions.AddEFCorePersistor(persistorOptions =>
                 {persistorOptions.ContextInstallers =
                                      new[] {
ModelConfig.EFCoreDbContextCreator }; });
    frameworkOptions.AddRestfulObjects(restOptions => {  });
    frameworkOptions.AddNakedObjects(appOptions => {
        appOptions.Types = ModelConfig.DomainModelTypes();
        appOptions.Services = ModelConfig.DomainModelServices();
    });
});
```

The template version shown above, delegates to methods defined on the `ModelConfig` class supplied in `Template.Model` project, the code for which is shown below:

```
public static class ModelConfig
{
    public static Type[] DomainModelTypes() => new[] { typeof(Student) };

    public static Type[] DomainModelServices() => new[]
{ typeof(ExampleService) };

    public static Type[] MainMenus() => new[] { typeof(ExampleService) };

    public static Func<IConfiguration, DbContext> EFCoreDbContextCreator =>
        c =>
        {
            var db = new ExampleDbContext(c.GetConnectionString("ExampleCS"));
            db.Create();
            return db;
        };
}
```

As shown above, you would need to add each new domain type, service, or menu into the code manually. However, it is also possible to write simple functions that use reflection to find and return, for example, all types with names matching a certain pattern, or within a certain namespace. Then you can add new domain types or services without having to update `ModelConfig`. For example:

```
private static Type[] DomainModelTypes =>
typeof(ModelConfig).Assembly.GetTypes().Where(t => t.IsPublic && (t.IsClass ||
t.IsInterface || t.IsEnum)).ToArray();
```

Note that as well as persistent domain entities, the list of domain types must include any view models, interfaces and enums that may need to be rendered on the user interface.

Configuring 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.

```
frameworkOptions.AddNakedObjects(appOptions => {
       ...
       appOptions.ConcurrencyCheck = true;
```

With this capability switched on, the client will need to provide the `if-match` header information. Domain objects will need to provide a suitable 'version' property, marked up with the `ConcurrencyCheck` attribute.

Configuring the EntityObjectStore

By default, the `Template.Server` project is configured to work with `EntityFrameworkCore`. It may also be configured to work with Entity Framework 6, by uncommenting the commented-out line shown below, and commenting-out the line below it:

```
services.AddNakedFramework(frameworkOptions => {
    frameworkOptions.MainMenus = MenuHelper.GenerateMenus(ModelConfig.MainMenus());
    // frameworkOptions.AddEF6Persistor(persistorOptions => { persistorOptions.ContextInstallers = new[]
{ ModelConfig. }; });
    frameworkOptions.AddEFCorePersistor(persistorOptions => { persistorOptions.ContextInstallers = new[]
{ ModelConfig.EFCoreDbContextCreator }; });
```

(You will need to add: `using NakedFramework.Persistor.EF6.Extensions;` and provide a new function – for example on `ModelConfig` – to create a `DbContext` that follows the EF6 patterns).

Configuring Authorization

(Read the section on Authorization first).

You will need create an instance of an implementation of `IAuthorizationConfiguration`. For example:

```
public static IAuthorizationConfiguration MyAuthConfig() {
    var config = new AuthorizationConfiguration<MyDefaultAuthorizer>();
    config.AddNamespaceAuthorizer<MyAppAuthorizer>("MyApp");
    config.AddNamespaceAuthorizer<MyCluster1Authorizer>("MyApp.MyCluster1");
    config.AddTypeAuthorizer<Bar, MyBarAuthorizer>();
    return config;
}
```

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

This authorization config must then be registered within the options for the `NakedFramework` for example:

```
services.AddNakedFramework(frameworkOptions => {
      ...
       frameworkOptions.AuthorizationConfiguration = MyAuthConfig();
```

Configuring Auditing

(Read the section on Auditing first).

You will need to create an instance of an implementation of `IAuditConfig`, for example:

```
private static IAuditConfiguration MyAuditConfig() {
    var config = new AuditConfiguration<MyDefaultAuditor>();
    config.AddNamespaceAuditor<MyAuditor1>("MySpace.Foo");
    config.AddNamespaceAuditor<MyAuditor2>("MySpace.Bar");
    return config;
}
```

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

This audit configuration must then be registered within the options for the `NakedFramework` for example:

```
services.AddNakedFramework(frameworkOptions => {
      ...
       frameworkOptions.AuditConfiguration = MyAuditConfig();
```

Configuring Profiling

(Read the section on Profiling first).

You need to configure your implementation of `IProfiler`, together with the set of events to be profiled, using an `IProfileConfiguration`, for example:

```
public static IProfileConfiguration MyProfileConfig() {
    var events = new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation };
//etc
    return new ProfileConfiguration<MyProfiler>() { EventsToProfile = events };
}
```

This profile configuration must then be registered within the options for the `NakedFramework` for example:

```
services.AddNakedFramework(frameworkOptions => {
      ...
       frameworkOptions.ProfileConfiguration = MyAuditConfig();
```

Configuring the user interface

See Configuring the user interface.

Configuring the Restful API

In Startup the highlighted line permits various options relting to the Restful API to be configured:

```
services.AddNakedFramework(frameworkOptions => {
        ...
         frameworkOptions.AddRestfulObjects(restOptions => {   });
```

Typing `restOptions.` in the empty braces will bring up a list of the options available with a documentary comment for each.

Configuring the RestRoot

If you wish to have a ‘root’ name that appears within any URL after the server name e.g.

```
[servername]/foo/objects/AdventureWorksModel.Customer/314
```

You can specify it in `Startup` here:

```
public void Configure(IApplicationBuilder app, ...
{
        ...
         app.UseRestfulObjects("foo");
```

How to determine whether an action will require a GET, PUT or POST method

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 specify 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*.

How to implement automatic redirection for specific objects

If a domain object implements the interface `NakedObjects.Redirect.IRedirectedObject` (defined in the `NakedObjects.Types.dll`) then when an attempt is made to retrieve the object (either directly via a link to the object resource, or as an action result), the server will return an http '301 Moved Permanently' to an alternative Restful Objects object resource that may even be on another server.

This pattern is particularly effective as a mechanism for integrating multiple existing systems. In the following example, the `Order` object may be thought of as existing in one system (‘`A`’) and the `Customer` in on another system (‘`B`’). However, system `A` has a very light-weight ‘stub class’ called `CustomerMemento`, which has minimal content, but implements `IRedirectedObject`.

The `Customer` class on the system B is in fact merely a 'stub' - the class is persisted on the new system (which is why it has its own `Id` property), but the persisted values for `ServerName` and `Oid` provide the necessary information for constructing a Url to the full `Customer` object on the system `B`. The `Title` property contains information that is duplicated from the data in the old system - to reduce the need for the user to navigate the link in order to see the identity of the customer – this is not required to implement `IRedirectedObject`. (Ideally, this title will be static identity information: if it were subject to change then it would be necessary to add a mechanism to enforce consistency).

```
//Stub object
public class Customer : IRedirectedObject    {

  public virtual int Id { get; set; }

  [Title]
  public virtual string Title { get; set; }

  public virtual string ServerName { get; set; }

  public virtual string Oid { get; set; }
}
```

Cache settings

When running a NOF application, there are, potentially, two quite different forms of caching within the client: Naked Objects client-managed caching, and browser-managed caching.

Naked Objects client-managed caching functions independently of the browser’s generic caching mechanism. This cache is *cleared whenever the user logs off, refreshes the page (using the browser’s refresh button), or closes the browser*. It relies both on the caching provided by `angular.js` and on bespoke Naked Objects caching functionality. These forms of caching provide the most effective performance improvements – for example in allowing you to back-track through the history of viewed objects without incurring server hits. These forms of caching do not pay any attention – deliberately – to the recommended caching periods provided by the server.

Browser-managed caching is is a generic feature of all browsers. Returned representations are cached for the length of time specified by the server in the header of each Http response, *even after the user has logged-off and/or closed the browser*.

The default cache settings are set to zero for all returned representations, effectively disabling the *browser-managed* caching. This is to avoid the issue where, for example, one user has logged off and another user has logged on from the same machine and browser. It might be that those users have different authorization levels and should perhaps then see a different set of main menus. If the browser has cached the menu representation then the second user might see the same main menus that were offered to the first user. Note that authorization is still enforced on the server – so the second user would not be able to invoke any action for which they were not properly authorized.

Given the effectiveness of Naked Objects client-managed caching, enabling browser-caching is unlikely to improve performance substantially. If you wish to enable browser-caching, this is done through the `CacheSettings` property of the `RestfulObjectsOptions`, for example:

```
builder.AddRestfulObjects(restOptions => {
        ...
         restOptions.CacheSettings = (2, 3600, 86400);
});
```

The 3-tuple specifies the *browser-managed* cache time (in seconds) for each of three different types of returned representation:

- Transactional representations: principally domain objects. Since these change frequently, zero-caching is recommended. However, in some high-volume systems, a caching value of 1-2 seconds can optimise performance.
- Short-term caching, for example of representations incorporating user-credentials.
- Long-term caching, used for representations that change rarely, usually involving a system redeployment: for example menus.

How to change the format of the Object Identifier (Oid) in resource URLs

Many of the resources (URLs) defined by the RESTfulAPI include an 'object identifier' in the form: `{DType}/{IID}`, where `{DType}`is the d*omain type identifier* and `{IID}` is the *instance identifier*; taken together these are referred to as the *object identifier* or *Oid*. By default, the Naked Objects 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, for example:

```
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 custom Authorization):

```
MyApp.Customers.WholesaleCustomer/f56bk-xx803h-jk4788ggweq2
```

Yet another reason would be if a domain object has one or more string keys that contain the default key separator. For example, if you have a natural (single) key of `sku--10452`, then the 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*.

Implementations of either or both types, should be registered as top-level services, for example:

```
Services.AddTransient<ITypeCodeMapper, MyTypeCodeMapper>();
Services.AddTransient<IKeyCodeMapper, MyKeyCodeMapper>();
```

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. If you wish to restrict the Restful API to a sub-section of the domain model, then there are several different techniques available to you:

- Use custom authorization to control visibility of classes, properties and actions to certain users or groups of users. Note that this approach could also be used to hide sections of the model from all users - without necessarily impacting any user interface

- because the two 'interfaces' can be generated by two different run projects, each of which can register a different custom authorizer.

- Make use of the `ITypeCodeMapper` and/or `IKeyCodeMapper` (see previous heading). While the primary intent of these interfaces is to allow control over the format of the object-identifier in resource URLs, they may 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 MyTypeCodeMapper : ITypeCodeMapper {

  public Type TypeFromCode(string code) {
    return IsExcluded(code) ? null : TypeUtils.GetType(code);
  }

  public string CodeFromType(Type type) {
    string typeName = TypeUtils.GetProxiedTypeFullName(type);
    return IsExcluded(typeName) ? null : typeName;
  }

  private bool IsExcluded(string typeName) {
    return typeName.ToUpper().StartsWith("MYAPP.PAYMENTS") ||
typeName.ToUpper().StartsWith("MYAPP.EMPLOYEES");
  }
}
```

Note the use of the two methods `TypeUtils.GetType` and `TypeUtils.GetProxiedTypeFullName`. These are the safest ways to convert between types and strings in Naked Objects. Calling `foo.GetType()` on a persistent object will return the Entity Framework proxied type, not the raw domain type - using `TypeUtils.GetProxiedTypeFullName` will get you the raw domain type.

Returning null from each of the methods when the intended type is to be excluded is sufficient to ensure that that type can never be accessed via the Restful API. A similar approach could be used with an implementation of `IKeyCodeMapper` to exclude certain ranges of *instance Ids* if desired.

Configuring cross-origin resource sharing (CORS)

This may be done within the `ConfigureServices` method on the `Startup` class within the server project, for example:

```
services.AddCors(corsOptions => {
    corsOptions.AddPolicy(MyAllowSpecificOrigins, policyBuilder => {
        policyBuilder
            .WithOrigins("http://localhost:5001", "http://localhost")
            .AllowAnyHeader()
            .WithExposedHeaders("Warning","ETag", "Set-Cookie")
            .AllowAnyMethod()
            .AllowCredentials();
    });
});
```

Note: If using **Internet Explorer** with CORS it is necessary to tweak the Advanced Cookie Settings to *Allow All.*

How to hook additional functionality into controller methods

Create a new 'filter'as a sub-class of the `ActionFilterAttribute`:

```
public class MyActionFilter : ActionFilterAttribute {
      public override void OnActionExecuting(HttpActionContext actionContext)
{
      //....my new functionality here
            var i = Guid.NewGuid();
        }

        public override void OnActionExecuted(HttpActionExecutedContext
actionExecutedContext) {
            //....and here
            var i = Guid.NewGuid();
        }
    }
}
```

Then add attribute to any methods on RestfulObjectsController that you require to extend e.g:

```
 [HttpGet]
 [MyActionFilter]
 public override HttpResponseMessage GetHome([ModelBinder(typeof
    (ReservedArgumentsBinder))] ReservedArguments arguments) {
    return base.GetHome(arguments);
}
```

the `HttpActionContext` gives access to the Request and to the Controller if required.

How to inject services or the Domain Object Container into the controller:

Use the `Inject` method on the `IFrameworkFacade`, which is passed into the constructor of the controller:

```
public IDomainObjectContainer Container { protected get; set; }

public RestfulObjectsController(IFrameworkFacade frameworkFacade) :
base(frameworkFacade) {
      frameworkFacade.Inject(this);
}
```

The Naked Objects Programming Model

Principal 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 this tells you all you need to know to create and deploy your own applications.

Domain object

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

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

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

You can specify certain things about both the behaviour and presentation of domain objects by adding specific attributes or methods. See 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. (Certain value types may provide alternative mechanisms for user input, such as a pop-up date-selector for a date field.)

Reference Property

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

Collection Property

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

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

You can specify certain things about both the behaviour and presentation of properties by adding specific attributes or methods. See Properties and 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 `NakedObjectsIgnore` attribute.

You can specify certain things about both the behaviour and presentation of actions by adding specific attributes or methods. See Actions.

See also: Contributed action.

Actions that return a collection

An action (on a service or a domain entity object) may return any of the recognised collection types. 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 be disabled.

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

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

Menus

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

Object Menus

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

```
public static void Menu(IMenu menu) {…}
```

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

```
public static void Menu(IMenu menu) {
    menu.AddAction("Action2");
    menu.AddAction("Action1");
    var sub1 =menu.CreateSubMenu("Sub1");
    sub1.AddAction("Action3");
    var sub2 = menu.CreateSubMenu ("Sub2");
    sub2.AddAction("Action4");
    menu.AddRemainingNativeActions();
    menu.AddContributedActions();
}
```

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

Note that the action names are specified in the same format as the method names in code - even if any of the actions have been given a customised name using the `Named` or `DisplayName` attributes.

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

```
menu.AddAction(nameof(Action2));
```

Sub-menu names should not contain underscores (e.g. `"My_sub_menu"`) – as this would cause them to be rendered incorrectly.

Main Menus

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

```
public static IMenu[] MainMenus(IMenuFactory factory) {
    var customerMenu = factory.NewMenu<CustomerRepository>();
    CustomerRepository.Menu(customerMenu);
    return new IMenu[] {
            customerMenu,
            factory.NewMenu<OrderRepository>(true),
            factory.NewMenu<ProductRepository>(true)
};
```

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

```
public static void Menu(IMenu menu) {
    menu.AddAction("FindCustomerByAccountNumber");
    menu.CreateSubMenu ("Stores")
        .AddAction("FindStoreByName")
        .AddAction("CreateNewStoreCustomer")
        .AddAction("RandomStore");
    menu.CreateSubMenu("Individuals")
        .AddAction("FindIndividualCustomerByName")
        .AddAction("CreateNewIndividualCustomer")
        .AddAction("RandomIndividual");
    menu.AddAction("CustomerDashboard");
}
```

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

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

```
public static void Menu(IMenu menu) {
    menu.WithName("My Hybrid Menu");
    menu.Type = typeof(ServiceA);
        .AddAction("ActionA1")
        .AddAction("ActionA2")
    menu.Type = typeof(ServiceB); // changing the type for the next additions
        .AddAction("ActionB1")
        .AddAction("ActionB2")
   .CreateSubMenu(("Cs").
       .Type = typeof(ServiceC)  // changing the type for the sub-menu now
        .AddAction("ActionC1")
   menu.AddAction("ActionB3")  // menu itself still has type ServiceB
}
```

Recognised method

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

Recognised attribute

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

View Model

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

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

IViewModel, IViewModelEdit, and IViewModelSwitchable

In Naked Objects, a View Model is coded the same way as a domain object but must implement one of these three interfaces: `IViewModel`, `IViewModelEdit`, or

`IViewModelSwitchable`. All three of these interfaces require that the following two methods are implemented:

- `DeriveKeys` must return a string array that defines the key(s) to this object. Typically these will be derived from the key(s) of the persistent object(s) that this view model represents, but could also directly represent value properties.

- `PopulateUsingKeys` takes the same array of keys as a parameter and is used to re-populate the view model whenever it is referenced in a request (either to refresh the view or to invoke an action on the view model, for example).

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

Important: the two methods *must be public*. This means that in C# the interface should be implemented conventionally and not 'explicitly'.

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

```
[NotMapped]  //Needed if working with EF Code First, so that no table is
created
public class CustomerComparison : IViewModel
{
    public IDomainObjectContainer Container { set; protected get; }  //Injected
service

    public virtual Customer Customer1 { get; set; }

    public virtual Customer Customer2 { get; set; }

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

    public void PopulateUsingKeys(string[] keys)
    {
        int id1 = int.Parse(keys.ElementAt(0));
        Customer1 = Container.Instances<Customer>().Single(x => x.Id == id1);
        int id2 = int.Parse(keys.ElementAt(1));
        Customer2 = Container.Instances<Customer>().Single(x => x.Id == id2);
    }

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

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

Note: The Restful Objects specification does not permit an instance identifiers (the simple or compound key) to contain any slashes - as this would create ambiguous URLs (see How to change the format of the Object Identifier (Oid) in resource URLs). If, for example, your View Model needs a date field as a key (in order that it can be re-created) then you should ensure that the date is translated into a string that does not contain any slashes, for example using:

```
var dateKey = myDate.ToString("ddMMyyyy");
```

Instantiating a view model

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

```
var cc = Container.NewViewModel<CustomerComparison>();
cc.Customer1 = customer1;
cc.Customer2 = customer2;
return cc;
```

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

```
[NotMapped]  //Needed if working with EF in Code First mode, so no table is
created
public class MyCustomerViewModel : ViewModel<Customer>
{
    //Properties (not shown), derive their information from the 'Root' property
(defined on ViewModel), which will be of type Customer, for example:

    public string Name {
        get {
            return Root.Name;
        }
    }
}
```

Distinctions between the three types of view model

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

Concurrency Checking and View Models

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

Service

Services perform three roles in a Naked Objects application:

- To provide methods for creating and retrieving domain objects where the user does not have an existing object to navigate from. Services that perform this role are often referred to as Factories & Repositories.

- To provide a bridge to external functionality. See 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. See Contributed Action.

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

Factories and Repositories

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

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

To write a repository you simply need to provide a write-only property into which the framework can inject an `IDomainObjectContainer`, 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. The following are examples of what we mean by bridging domains:

- Linking to functionality that already exists, or has to exist, outside of the Naked Objects application, such as pre-existing services, or functionality within legacy systems.

- Bridging between technical domains, such as between the object domain and the relational database domain, or the email domain.

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

```
public class SmtpMailSender : AbstractService, IEmailSender
{
    private static string SMTP_HOST_NAME = "localhost";
    private static string SMTP_USER = "admin@example.com";

    public void SendTextEmail(string toEmailAddress, string text)
    {
        SmtpClient client = new SmtpClient();
        client.Host = SMTP_HOST_NAME;

        MailMessage message = new MailMessage();
        message.Sender = new MailAddress(SMTP_USER);
        message.From = new MailAddress(SMTP_USER);
        message.To.Add(new MailAddress(toEmailAddress));
        message.Subject = "Expenses notification";
        message.Body = text;

        client.Send(message);
    }
}
```

Contributed actions, properties or collections

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

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

Actions contributed to individual objects

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

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

```
public class TaskContributedActions : AbstractService
{
    public Task CreateNewTask([ContributedAction("Tasks")] ITaskContext
context)
    {
        var task = Container.NewTransientInstance<Task>;
        task.Context = context;
        return task;
    }
    //...
}
```

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

Actions contributed to query results (‘query-contributed actions’)

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

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

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

```
public void MarkAsReconciled(
         [ContributedAction()] IQueryable<Transaction> transactions) {

    foreach (Transaction t in transactions.ToList())
    {
        t.MarkAsReconciled();
    }
}
```

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

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

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

Contributed properties or collections

It is possible to contribute a read-only property or a collection to any object of a given type, using `DisplayAsProperty`:

```
public class TaskContributedProperties : AbstractService
{
    [DisplayAsProperty]
    public ICollection<Task> CurrentTasks([ContributedAction("Tasks")]
ITaskContext context)
    {
        //...
    }
}
```

Injection of domain services into domain objects

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

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

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

```
public class Product {
    public string Colour { }

    public ProductRepository ProductRepository { set; protected get; }

    public IQueryable<Product> FindOtherProductsOfSameColour() {
        return ProductRepository.ListProductsForColour(this.Colour);
    }
}
```

Tip: Use the `injs` snippet to create this code.

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

```
public class Product {
    public IPricingMechanism[] PricingMechanisms { set; protected get; }

    public void CalculateBestPrice() {
        this.Price = PricingMechanisms.Min(pm => pm.Price(this);
    }
}
```

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

```
public IPricingMechanism PricingMechanism { set; protected get; }
```

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

The Domain Object Container

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

This is done via the domain object container, which is defined by the interface `NakedObjects.IDomainObjectContainer`.

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

```
public IDomainObjectContainer Container { set; protected get; }
```

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

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

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

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

IDomainObjectContainer methods

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

- `NewTransientInstance<T>` returns a new object of a specified type in a transient (not yet persistent) state. If this transient object is returned to the user then it will appear in edit mode ready to be completed and then saved with the Save button. Alternatively, the programmer may persist the transient object with ...

- `Persist(ref object).` Note that the object must be passed into the method 'by reference'. This is because the object will be replaced by a new object, with the same state, but now managed by the Entity Framework. An object is only ever persisted once - thereafter it remains in a persistent state (and is managed by the object Persistor) and its properties are updated as needed. Calling `Persist` on an object that is already persistent will throw an error.

- `IsPersistent(object)` allows you to check if a particular object is already persistent.

- `DisposeInstance(object)`allows you to delete a persistent object. The programmer must take responsibility to ensure that any references to this object from other associated objects are cleared as part of the same transaction.

- `Instances` returns all the instances of a specified type as an `IQueryable`, upon which LINQ queries can then be performed. There are two overloaded versions of this method: one is templated and should be used wherever the type of object being retrieved is known statically. The other takes a `System.Type` as a parameter and may be used where the type is not known statically.

- `InformUser(string)` sends an informational message to the user.

- `WarnUser(string)` sends a warning message to the user.

- `RaiseError(string)` sends an error message to the user.

- `Resolve(object)` ensures that an object has been fully resolved from the Persistor. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework. **Note:** `Resolve` and `ObjectChanged` (below) are largely a hangover from older versions of Naked Objects and might be removed from the interface in a future release.

- `ObjectChanged(object)` is used to notify Naked Objects that an object has changed and that those changes may need to be notified to the object Persistor and/or to the user interface. Note that there is no need to call this method explicitly in a typical Naked Objects application since it is called automatically by the proxy object that is created behind-the-scenes by the Entity Framework.

- `Principal` (property) returns a `System.Security.Principal.IPrincipal`, from which you may obtain the user's identity and thence the name.

- `AbortCurrentTransaction`. Any uncaught exception thrown within an action method will automatically abort the transaction. If you wish to abort a transaction without throwing an exception you may use this method.

- `NewTitleBuilder()` (plus two overloads). Returns an implementation of ITitleBuilder, which provides convenience methods for building object titles from properties and references.

- `TitleOf(object)` As an alternative to using `NewTitleBuilder`, this method provides a convenient way to obtain the title of another domain object, without having to know how that title is defined.

- `Refresh(object).` Ensures that the object is updated with the latest values from the object store.

- `NewViewModel<T>.` Broadly equivalent to `NewTransientInstance`, but for View Models (which are never persisted).

- `GetService<T>.` Mechanism to obtain a specific service other than by dependency injection.

Recognised types, attributes, and conventions

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

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

Recognised Value Types

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

- `System.Boolean`

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

Recognised Collection types

Collections are recognised by Naked Objects in two different contexts:

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

2. As a property on on object, representing a multiple association

These are defined below.

Collections returned by an action

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

```
public IEnumerable<Customer> Xxx() {}
public IQueryable<IPerson> Xxx() {}
public ICollection<IDocument> Xxx() {}
public IList<Employee> Xxx() {}
public SalesOrder[] Xxx() {}
```

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

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

```
public IEnumerable<decimal> Xxx() {}
public IQueryable<DateTime> Xxx() {}
public ICollection<int> Xxx() {}
public IList<IComparable> Xxx() {}
public String[] Xxx() {}
```

Collections as properties on an object

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

```
public ICollection<IDocument> Xxx {get{} set{}}
public IList<Employee> Xxx {get{} set{}}
public SalesOrder[] Xxx {get{} set{}}
```

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

Recognised .NET attributes

Naked Objects recognises the following .NET attributes.

ConcurrencyCheck

`System.ComponentModel.DataAnnotations.ConcurrencyCheck`

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

ComplexType

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

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

```
public class Person {

    public virtual Name FullName { get; set; }
}

[ComplexType]
public class Name {

    public virtual string FirstName { get; set; }
    public virtual string LastName { get; set; }
}
```

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

(See also the `Root` attribute).

DataType

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

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

DefaultValue

`System.ComponentModel.DefaultValue`

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

```
public class Invoice
{
   [DefaultValue(30)]
  public virtual int CreditDays { get; set; }
}
```

Description

`System.ComponentModel.Description`

May be used as an alternative to the `NakedObjects.DescribedAs` attribute.

DisplayName

`System.ComponentModel.DisplayName`

May be used as an alternative to the `NakedObjects.Named` atttribute.

MaxLength

`System.ComponentModel.DataAnnotations.MaxLength`

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; }
}
```

MinLength

`System.ComponentModel.DataAnnotations.MinLength`

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

Range

`System.ComponentModel.DataAnnotations.Range`

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

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

RegularExpression

`System.ComponentModel.DataAnnotations.RegularExpression`

This may be used as an alternative to the `RegEx` attribute.

ScaffoldColumn

`System.ComponentModel.DataAnnotations.ScaffoldColumn`

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

StringLength

`System.ComponentModel.DataAnnotations.StringLength`

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

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

Recognised Methods

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

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

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

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

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

- Complementary methods.
- LifeCycle methods
- Other recognised methods.

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

Complementary methods

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

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

- `AutoComplete`: Used to generate a dynamic drop-down list of matching objects or values. See How to specify auto-complete for a property and How to specify auto-complete for a parameter.

- `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. SeeHow to specify a set of choices for a property and How to specify a set of choices for a parameter .

- `Default`: A complementary method used in conjunction with a property or an action. See How to specify a default value for a property and How to specify a default value for a parameter.

- `Disable`: A complementary method used in conjunction with a property or an action. The `Disable` dynamically controls whether a field is editable, or an action can be initiated. If a `String` is returned the field or action is disabled and the string is made visible to user to inform them why it is disabled. If the method returns `null` then the field or action remains enabled. See How to prevent the user from modifying a property and Disabling an action based on the state of the object.

- `Hide`: A complementary method used in conjunction with a property or an action. The `Hide` method allows a property or action to be dynamically hidden from the user, such as hiding a field once it is set up. Returning `true` makes the property or action invisible. See How to hide a property from the user and How to hide actions.

- `Modify`: A complementary method used in conjunction with a property. The `Modify` method is called when the user (rather than the framework) sets a reference or value field. This is typically used to trigger other behaviours such as updating a total. See How to trigger other behaviour when a property is changed.

- `Validate`: A complementary method used in conjunction with a property or an action to validate value(s) entered by the user. SeeHow to validate user input to a property and How to validate parameter values.

LifeCycle methods

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

- `Created`: Life cycle method called by framework when an object is first created. This is the instance's logical creation. This method will *not* be called when the object is retrieved from persistent storage into memory.

- `Deleted`: Life cycle method called by framework when an object has just been removed from the persistent store. At this point the object will exist in memory, but no longer exist in the persistent store.

- `Deleting`: Life cycle method called by framework when an object is just about to be removed from the persistent store. At this point the object still exists in the persistent store.

- `Loaded`: Life cycle method called by framework when an object has just been loaded into memory from the persistent store. At this point the object has had its state fully restored. `Loaded` will be called after the object has been loaded and before the transaction has completed. When retrieving an object via the user interface this means that `Loaded` will have been called by the time the object appears on the screen. However, if you are processing objects programmatically - whether from within a user action or from an external call - then be aware that the `Loaded` might not be called on any (or all) of the objects being processed until the very end of the transaction. So if your method involves loading objects and processing them, you cannot assume that `Loaded` will have been called before you get hold of each object. In general, it is recommended that you use `Loaded` only for very simple, non-invasive purposes, such as calculating a total for display purposes before an object is returned to the user.

- `Loading`: Life cycle method called by framework when an object is just about to be loaded from the persistent store. At this point the object exists in memory but has not had its state restored. You should never attempt to reference any non-scalar property within your `Loading` method. Unlike the other pairs of methods, there may be a considerable time gap between the calling of `Loading()` and `Loaded()` on an object - with the latter being called only when the object is first displayed or used programmatically. For most application purposes, `Loaded()` is a more useful event than `Loading()`.

- `OnPersistingError`: Life cycle method called by the framework if the object persistor throws an exception when an object is persisted. Typically this will be a `DataUpdateException` or an `OptimisticConcurrencyException`. By adding the `OnPersistingError` method to your code you can intercept this exception and parse its message to establish details. When the `OnPersistingError` method exits, the framework will still throw an exception to be caught by the user interface. However, you may specify the message for that exception by returning the desired message as a string from your method. There is no point in adding this method to your code unless you want to change the message that is passed to the user - otherwise just leave the exception handling to the framework. Note that if your returned string message contains one or more line-breaks (`\n`) then any text after the first line-break will be moved into the `Details` section of the resulting dialog.

- `OnUpdatingError`: Life cycle method called by the framework if the object persistor throws an exception when an object is being updated. Works in a similar manner to `OnPersistingError`.

- `Persisted`: Life cycle method called by framework *after* a transient object has been persisted. **Important**: unlike `Persisting`, the `Persisted` method will be in a separate transaction to the persisting of the object. This is useful because it means, for example, that if the key is database-generated then that generated value will be visible from within the `Persisted` method, but not in the `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

- `Title`: A `Title` method, returning a string, is one way to specify the title for an object. See How to specify a title for an object .

- `ToString`: If no title attribute, or title method (see above) has been specified, then the framework will call the object's `ToString` method to get a title for the object.

- `Menu.` Method to define a menu on an object. SeeObject Menus .

- `IconName.` This is a recognised method, so will not show up as an action. However, it no longer serves any valid purpose and will be removed in a future release.

NakedObjects.Attributes

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

AuthorizeAction

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

AuthorizeProperty

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

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 standard user-interface uses this information to render all the instances of this class available to the user as a drop-down list.

ContributedAction

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

DescribedAs

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

```
public class Customer {
  [DescribedAs("Approximate age (years only)")
  public virtual int Age() {get; set;}

  [DescribedAs("Create a new order object")
  public Order Order() {…}
}
```

Disabled

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

```
public class Customer {

  [Disabled]
  public virtual Money InitialCreditRating {...}
}
```

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

```
public class Customer {
  [Disabled(WhenTo.UntilPersisted)]
  public virtual Money InitialCreditRating {...}
}
```

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

DisplayAsProperty

Any side-effect-free method that returns a type that could be displayed as a property (including a collection) may be displayed as a property on an object, by annotating it with `DisplayAsProperty`. This applies whether that method is defined as an instance method on the object type (or supertype), or as a contributed action.

Eagerly

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

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

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

The syntax for the `Eagerly` attribute is as follows. Note that to use the `Do` enum you will need an `import NakedFramework;`

```
[Eagerly(Do.Render)]
public Customer Customer {get; set;}
```

Edit

Where the purpose of an action is to edit one or more properties on an object, annotating it with `[Edit]` will change how it is rendered on the default client. Instead of appearing in the object's action menu, an 'edit' icon will now appear next to each of the fields corresponding to parameter(s) of the method. And instead of generating a separate dialog, as the menu action would, clicking on any edit icon will cause the field(s) to become editable. Notes:

- Each parameter of a method marked up with [`Edit]` must match a property on the object type, both in type and name (except for casing).
- Each property on an object type may only be featured in, at most, one `[Edit]` method.
- When an edit icon is clicked, if the corresponding `[Edit]` method applies to more than one property, each of the properties will be editable, but just one **OK/Cancel** button will be rendered – after the last editable field.
- `Validation`, `Default`, `Choices`, and `AutoComplete` 'complementary' methods will work the same as if the method was rendered as an object-menu action.
- `[Edit]` is recognised only for actions defined on the object type (or supertype). It will not be recognised for contributed actions.

FinderAction

This attribute no longer has any direct significance for the default user interface, but it is included in the RESTful API for use in customised user interfaces.

FindMenu

This attribute now has no effect, but is being kept for backwards compatibility.

Hidden

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

```
public class Customer {
  [Hidden(WhenTo.OncePersisted)]
  public virtual int InternalId { get; set; }
}
```

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

Idempotent

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

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

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

(See also the `QueryOnly` attribute).

Immutable

The Immutable attribute may be applied to a class, to indicate that the state (properties) of a an object may not be modified by the user - either by editing and saving, or by invoking an action on the object that might modify those properties.

(However, it is acceptable to invoke ‘query-only’ actions on an object - meaning those actions that return an `IQueryable` or that are explicitly annoted with `[QueryOnly]`).

```
[Immutable]
public class Country {...}
```

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

```
[Immutable(WhenTo.OncePersisted)]
public class Country {...}
```

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

Mask

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

Note that if you may also apply the `Mask` attribute to a reference property. However, to ensure that the mask is then applied to the title of the referenced object, you must ensure that the referenced object type has an overridden parameterless `ToString` method, *and* a `ToString` method that takes the mask as a parameter.

MemberOrder

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

```
[MemberOrder(<relative position>)]
```

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

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

```
public class Customer {
  [MemberOrder(2.1)]
  public virtual string Address {get; set;}

  [MemberOrder(1.1)]
  public virtual string FirstName {get; set;}

  [MemberOrder(1.2)]
  public virtual string LastName {get; set;}

  [MemberOrder(3)]
  public virtual Date DateOfBirth {get; set;}
}
```

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

MultiLine

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

```
public class BugReport {
  [MultiLine(NumberOfLines = 10)]
  public virtual string StepsToReproduce() { get; set; }
}
```

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

As of NOF 8.1, MultiLine may also be added to an action to indicate that the action should be rendered to allow multiple invocations with multiple sets of parameters i.e. like a data grid. See How to create an action that will allow multiple rows of data to be entered.

NakedObjectsIgnore

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

Named

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

NotPersisted

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

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

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

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

Optionally

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

PageSize

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

Plural

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

```
[Plural("Children")]
public class Child {...}
```

PresentationHint

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

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

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

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

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

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

In the standard client, these presentation hints are reproduced in the `class` attribute of the corresponding Html member.

QueryOnly

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

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

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

(See also the Idempotent attribute).

RegEx

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

```
[RegEx(Validation = <regEx string>)]
```

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

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

```
public class Contact {
  [RegEx(Validation = @"^[\-\w\.]+@[\-\w\.]+\.[A-Za-z]+$")]
  public virtual string Email { get; set; }
}
```

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

Root

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

```
[ComplexType]
public class Name {
    [Root]
    public virtual Person Person { get; set; }

    public virtual string FirstName { get; set; }

    public virtual string LastName { get; set; }
}
```

TableView

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

```
[TableView(true, "Product", "OrderQty", "UnitPrice")]
public virtual ICollection<SalesOrderDetail> Details {...}

[TableView(false, nameof(Task.Description), nameof(Task.DueDate))]
public IQueryable<Task> MyTasks() {...}
```

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

Note the following:

- The columns should take the same format as the property name in the code, not as displayed to the user: `"OrderQty"` not `"Order Qty"` - the column headers will automatically pick up the correct display format (or the overridden display name if one has been specified). For this reason we strongly recommend using the `nameof` keyword introduced in C# 6.0, as in the second example above.

- `[TableView(true)]` will result in a single-column table with just the title of the object, equivalent to a 'list view'.

Title

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

ValidateProgrammaticUpdates

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

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

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

NakedObjects.Types

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

NakedObjects

- `DomainException`. All exceptions that are generated within the application domain code should inherit from this class or throw it directly as this allows the Naked Objects Framework to discriminate between potential framework errors, infrastructure errors, and exceptions raised in application code.

- `IDomainObjectContainer`. See IDomainObjectContainer methods.

- `IViewModel`, `IViewModelEdit`, `IViewModelSwitchable`. See View Model.

- `ITitleBuilder`. See How to specify a title for an object.

NakedObjects.Async

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

NakedObjects.Audit

- `IAuditor`. Allows domain programmers to define an auditing service. See Configuring Auditing.

- `INamespaceAuditor`. Specific sub-type of `IAuditor`, where the methods will only be called in relation to types that fall within the specified namespace.

NakedObjects.Menu

- `IMenu`. See Object Menus.

NakedObjects.Profile

- `IProfiler`. See Profiling.
- `ProfileEvent` an enum defining the various events that may be detected for profiling.

NakedObjects.Redirect

- `IRedirected`. Implemented by a 'stub' class that acts as a proxy for an object managed on another server.

- `IRedirectedObject` and `IRedirectedService`.

NakedObjects.Security

- `INamespaceAuthorizer`. An implementation of this interface provides authorization for a single fully-qualified type, or for any types within a namespace. See Custom Authorization.

- `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.

- `Image` now provides the same functionality to `FileAttachment` (in prior versions of Naked Objects there were differences) See How to display an image.

- `IStreamResource`. Interface implemented by both `FileAttachment` and `Image`. (Not intended to be used directly within domain code.)

NakedObjects.Helpers

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

NakedObjects

- `DateTimeExtensions`. Useful methods that will automatically be added to instances of `DateTime` for comparing dates, deliberately ignoring the time element, for example: `IsBeforeToday()`

- `ViewModel`. A ready-made implementation of `IViewModel`, for view models based on a single Root type. See View Model.

- `IHasGuid`. This interface (along with several others below) is intended for use with 'Polymorphic Associations' (See How to handle associations that are defined by an interface rather than a class) and specifically to work in conjunction with `NakedObjects.Services.ObjectFinder`. If the class being associated implements `IHasGuid`, then the compound key will use this Guid (together with the fully qualified type name) to form the compound key. This has the advantage that the Guid may be set up when the object is created rather than waiting until the object is persisted (if the keys are database generated, that is). This is important when defining interface associations between transient objects that are all persisted in one transaction.

- `IHasIntegerId`. Merely defines that object has a single integer key called `Id`. Used by `PolymorphicNavigator`, for example.

- `IPolymorphicLink`. Designed to work with the `PolymorphicNavigator` service (below).

- `PolymorphicLink`. A ready-made implementation of `IPolymorphicLink`.

- `ReasonBuilder`. Helper object for constructing a string reason to be rendered to the user within, say, a Validate or Disable method. Separates multiple appended reasons with semi-colons.

NakedObjectsServices

- `AbstractFactoryAndRepository`. Convenience super class for factories and repositories that wish to interact with the container.

- `IKeyCodeMapper`. Defines a service that can convert between a key and a string code. Possible uses include: key encryption, custom key separators.

- `IObjectFinder`. Defines a mechanism for retrieving a domain object given a 'compound key' (a single string that defines both the type and the identity of that object). Also provides a method for determining the compound key for a given object. The intent of making both methods templated, is that it allows for the possibility of different types (or perhaps different namespaces of types) having different ways of putting together the compound key.

- `ITypeCodeMapper`. Defines a service that can convert between a Type and a string code where you don't wish to use the fully-qualified type name as the string representation. Possible uses include: to create compound keys for defining polymorphic associations;To create Oids for use in URLs.

- `ObjectFinder`. An implementation of `IObjectFinder`. Works with multiple keys, of type Integer, String, Short, or Char

- `ObjectFinderWithTypeCodeMapper`. An implementation of `IObjectFinder` that will delegate the string representation of a type to an injected `ITypeCodeMapper` service, if one exists. (Otherwise it will default to using the fully-qualified class name).

- `PolymorphicNavigator`. See How to handle associations that are defined by an interface rather than a class.

- `PolymorphicNavigatorWithOid`. An alternative implementation of the PolymorphicNavigator designed to work specifically with implementations of `IPolymorphicLinkWithOid`.

- `SimpleRepository`. This is a simple, typed, implementation of a repository that can be useful when writing tests.

NakedObjects.Utils

- `KeyUtils`. Utility methods for obtaining and making use of domain object keys, whether explicitly defined, or inferred by convention.

- `NameUtils`. Utility methods for manipulating names of domain model elements - for use in presentation, for example.

- `TypeUtils`. Utility methods for safely obtaining and using types defined within a domain model.

Using the Naked Objects Snippets

The Naked Objects source code on GitHub includes a number of C# Code Snippets that may be installed into Visual Studio. The snippets may be found here:

https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Programming%20Model/C%23%20Code%20Snippets

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.
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.

Name	Shortcut	Description
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`	See Polymorphic Association
Polymorphic Collection	`polycoll`	See Polymorphic Association
Result Interface Association	`resultia`	See Result interface association

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.

A how-to guide

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.

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.

Note: You should not attempt to add a reference to a transient object into another object (whether the second object is itself transient or persisted) until the first object has been persisted, as described below.

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 `Persist` method on the Domain Object Container. For example:

```
Customer newCust = Container.NewTransientInstance<Customer>;
newCust.Name = "Charlie";
Container.Persist<Customer>(ref newCust);
```

Note: The `Persist` method must have the (transient) object to be persisted passed in by reference (`ref`) – because it will be replaced by a new (persisted) object of the same type and with the same values.

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.

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 you have written a suitable method 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.

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

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

How to specify that an object should never be persisted

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

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

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

Use the `Immutable` attribute.

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

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

How to handle concurrency checking

Naked Objects provides full support for concurrency checking - such that before a user saves any edits, or invokes any action upon an object, the framework will check to see that no other user has changed the state of that object.

Any domain object that needs to make use of this capability, must have a 'version' property that is guaranteed to change each time the persisted state of that object is changed. This property may be of type `DateTime` (acting as a time stamp), a `string`, a numeric value or a byte array. It may be visible to, or hidden from, the user. The responsibility for updating the property when changes are persisted may be performed by the domain code, but will more commonly be performed by the database, by means of a trigger or a calculated column.

The property must be marked up with the `ConcurrencyCheck` attribute, as shown in the example below:

```
public class Employee {
     ...

   [ConcurrencyCheck]
   public virtual DateTime LastUpdated { get; set; }
}
```

If you have any inheritance within your domain model, then the `ConcurrencyCheck` attribute should be applied to a property on the top-most class of each hierarchy, and should not be duplicated within any sub-classes. (Sub-classes may have their own `LastUpdated` or similar properties for other purposes, but these do not play a role in concurrency checking.)

Note that there is no automatic concurrency checking on objects that are used as parameters for actions, nor on objects selected to be invoked by a query-contributed action.

Note: As of NOF 8, concurrency checking is not undertaken on the objects selected to be invoked with a 'query-contributed action'. This change was to make the behaviour consistent with other aspects of the system. For example, if an object is used as a parameter within an action there is no automatic concurrency checking on that object.

Object presentation

How to specify a title for an object

A title is used to identify an object in the user interface. For example, a `Customer` object'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`.

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

```
public string Title()
{
  var t = Container.NewTitleBuilder();
  t.Append(FirstName).Append(LastName).Append("," DateOfBirth, "dd-MMM-yy");
  return t.ToString();
}
```

would produce a title of the form:

```
Joe Bloggs, 07-Jan-63
```

Use the Title snippet for creating this method.

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

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

How to control the layout of an object's action menu

See Object Menus.

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 a formatted version of 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.

The programmer may optionally also provide a `DescribedAs` attribute.

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 declaratively, use the `Disabled` attribute.

To disable a property dynamically (for example, depending upon the status of the object as defined in other properties or methods) 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.

Note: If all properties on an object are disabled, the user will not be presented with the option to edit the object.

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

Use the Optionally attribute.

How to specify the size of String properties

Use the StringLength, and MultiLine attributes.

How to validate user input to a property

To validate that an input falls within a specific range, use the Range attribute.

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>(<type> value)
```

If the proffered value is deemed to be invalid then the property will not be changed. This is done by returning a non-empty String that 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;
  }
}
```

If the method returns null, or an empty string, then the values proffered are deemed to be valid.

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

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

```
public string ValidateMark(int mark){
  ReasonBuilder rb;
  rb.AppendOnCondition(! (mark >= 0 && mark <= 30), "Mark must be in range 0 to
30");
  return rb.Reason;
}
```

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

How to validate user input to more than one property

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

```
public virtual DateTime FromDate { get; set; }

public string ValidateFromDate(DateTime d) {
  if (!d.IsAfterToday()) {
    return "Must be after Today";
  }
return null;
}

public virtual DateTime ToDate { get; set; }

public string Validate(DateTime fromDate, DateTime toDate) {
  if (fromDate.Date > toDate.Date) {
     return "From Date cannot be after To Date";
  }
  return null;
}
```

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

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

How to specify a default value for a property

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

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

```
public <Property type> Default<PropertyName>()
```

For example:

```
public class Order
{
  public virtual Address ShippingAddress() Address { get; set;}

  public Address DefaultShippingAddress()
  {
    return Customer().NormalAddress();
  }
}
```

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

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

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

How to specify a set of choices for a property

See also How to handle enum properties andHow 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).

How to specify auto-complete for a property

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

```
public virtual Customer ForCustomer { get; set; }

[PageSize(10)]
public IQueryable<Customer> AutoCompleteForCustomer( [MinLength(3)] string
name) {
  return CustomerRepository.FindCustomerByName(name);
}
```

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

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

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

Note: For a reference property, the return type of the AutoComplete method must be either IQueryable<T> where T is the type of the property, or just T (i.e. a single matching object). For a string property only, the return type of the AutoComplete method may be IEnumerable<string> - in which case PageSize is not applicable.

How to set up the initial value of a property programmatically

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

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 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.

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.

This description will appear as a 'placeholder' (greyed-out text) within the input field when the property is being edited *but only when the field is empty and the focus is outside the field.*

How to hide a property from the user

To hide a property declaratively use the `Hidden` attribute.

To hide a property base on the state of the object use a `Hide` method. The syntax is:

```
public bool Hide<PropertyName>()
```

Returning `true` 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 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 - 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 `System.ComponentModel.DataAnnotations.Schema.NotMapped` attribute.

Note that there is also the option to render the result of a side-effect-free method as a property using `DisplayAsProperty`:

```
public class Employee
{
  public virtual Department Department { get; set;}

  [DisplayAsProperty]
  public Employee Manager() => Department is null? null : Department.Manager();
  ...
}
```

How to handle File Attachments

The pattern for allowing the user to attach a file is shown in the following example code:

```
public virtual FileAttachment Attachment
{
    get
    {
        if (AttContent == null) return null;
        return new FileAttachment(AttContent, AttName, AttMime);
    }
}

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

[NakedObjectsIgnore]
public virtual string AttName { get; set; }

[NakedObjectsIgnore]
public virtual string AttMime { get; set; }

public void AddOrChangeAttachment(FileAttachment newAttachment)
{
    AttContent = newAttachment.GetResourceAsByteArray();
    AttName = newAttachment.Name;
    AttMime = newAttachment.MimeType;
}

//Alternatively:
//public void AddOrChangeAttachment(FileAttachment newAttachment, string
withNewName)
//{
//    AttContent = newAttachment.GetResourceAsByteArray();
//    AttName = withNewName;
//    AttMime = newAttachment.MimeType;
//}
```

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

In-line images are handled in the same way as file attachments. (The `NakedObjects.Value.Image` type will work as an alternative to `FileAttachment`. In previous versions of Naked Objects there was a difference between the way these two were handled. As of NOF9 there is no distinction).

If the mime type of a file attachment is an image then a thumbnail of the image will be shown inline within the object. You can clicl on it to display the image standalone - but still with the icon bar at the bottom to navigate back to where you were in the application - or you can right-click on the image to display it in the other pane alongside the object.

Note: It is not currently possible to create an image within a transient object - it can only be added once the object has been persisted.

How to handle enum properties

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

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

```
public Sexes Sex {get; set;}
...
public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}
```

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

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

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

```
[EnumDataType(typeof(Sexes))]
public int Sex {get; set;}
...
public enum Sexes {Male=1, Female=2, Unknown=3, NotSpecified=4}
```

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

How to work with date properties

(As of NOF8/9) A property of type `DateTime` in the domain code will be represented on the user interface according to the following rules.

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

Collection properties

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

How to add a collection property to a domain object

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

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

```
public class Order {
  ...
  private ICollection<OrderLine> myLines = new List<OrderLine>();

  public virtual ICollection<OrderLine> Lines  {
    get {
      return myLines;
    }
    set {
      myLines = value;
    }
  }
}
```

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

Adding-to or removing objects from a collection

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

```
  public class Customer {

       #region Vehicles (collection)
       private ICollection<Vehicle> _Vehicles = new List<Vehicle>();

       public virtual ICollection<Vehicle> Vehicles {
           get {
               return _Vehicles;
           }
           set  {
               _Vehicles = value;
           }
       }

       public virtual void AddToVehicles(Vehicle value) {
           if (!(_Vehicles.Contains(value)))  {
               _Vehicles.Add(value);
           }
       }

       public virtual void RemoveFromVehicles(Vehicle value) {
           if (_Vehicles.Contains(value)) {
               _Vehicles.Remove(value);
           }
       }

       public IList<Vehicle> ChoicesRemoveFromVehicles(Vehicle value) {
           return Vehicles.ToList();
       }
       #endregion
   }
```

Note the following:

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

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

How to create a derived collection

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

```
public class Department
{
  // Derived collection
  [NotPersisted, NotMapped]
  public ICollection<Employee> TerminatedEmployees
  {
    get
    {
      List<Employee> results = new List<Employee>();
      foreach (Employee e in Employees)
      {
        if (e.IsTerminated())
        {
          results.Add(e);
        }
      }
      return results;
    }
  }
  ...
}
```

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

How to control the order in which table rows are displayed

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

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

public virtual ICollection<SalesOrderDetail> Details {
  get {
    return _details.OrderBy((x) => x.UnitPrice).ToList();
  }
  set {
    _details = value;
  }
}
```

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

How to specify which columns are displayed in a table view

Use the `TableView` attribute.

How to create an action that operates on a selection from a collection

Note: this section refers only to collections within an object ('multiple associations'). For actions that operate on a selection from a list of objects returned as the result of a query action, see query-contributed actions.

It is possible to define an action on an object that may be applied to any selection of members within a specified collection belonging to that object. To do this:

- the first (or only) parameter of the action must be of type `IEnumerable<T>,` where `T` is the type of the collection
- the name of that parameter must match the name of the collection (though with the first character being lower-case)
- that parameter must be marked up with `[ContributedAction]`

Example:

```
public class OrderHeader
{
    private ICollection<OrderDetail> details = new List<OrderDetail>();

    public virtual ICollection<OrderDetail> Details
    {
        get { return details; }
        set { details = value; }
    }

    public void RemoveDetails([ContributedAction] IEnumerable<OrderDetail>
details)
    {
        foreach (OrderDetail detail in details)
        {
            if (Details.Contains(detail))
            {
                Details.Remove(detail);
            }
        }
    }
}
```

Adopting this pattern will mean that:

- The `Details` collection, when opened on the UI, will have a check box at the start of each row
- The action `Remove Details` will appear as a button within the *opened* collection
- The user may select one, multiple, or all of the rows and then invoke the action by hitting that button

Note also that:

- Such actions may also take additional parameters, in which case a dialog will be opened immediately above the collection.
- Such actions may have the usual range of complimentary methods to perform validation, disable the method, or provide auto-complete on parameters (except for the `IEnumerable` parameter, which is set only by selecting rows from the collection) for example.

See also: How to make an action appear within an object collection.

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.

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

See Object Menus.

How to define a contributed action

See 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.

How 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.

Note: Due to the metadata representations used internally by Naked Objects, an action may not have a parameter that is simply the word `action` – this will result in an error message such as:

```
The UriTemplate 'services/{oid}/actions/{action}/invoke?action={action}' is not
valid; the UriTemplate variable named 'action' appears multiple times in the
template
```

However, the same effect can be achieved by using the `DisplayName` attribute e.g:

```
  public void Foo([DisplayName("Action")] int bar)
```

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

How to make a parameter optional

Use the `Optionally` attribute.

How to permanently disable a parameter optional

The `Disabled` attribute may be added to an action parameter, such that the user may see the name and contents of the parameter but may not edit the latter. This makes sense only if a default value has been provided for the parameter (see below).

An example of when this might be useful would be to use a disabled parameter to provide the user with a reminder of a relevant value from a previous context, or simply to provide explanatory text visible within the dialog.

How to specify a default value for a parameter

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

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

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

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

```
public <parameter type> Default<ActionName>([<parameter type>
parameterNameSameAsOnAction])
//OR
public <parameter type> Default<parameter number><ActionName>()
```

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

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

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

  public Product DefaultPlaceOrder(Product product)
  {
    return ProductMostRecentlyOrderedBy(this.getCustomer());
  }

  public int Default1PlaceOrder()
  {
    return GetQuantityFromPreviousOrder();
  }
  ...
}
```

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

How to specify a set of choices for a parameter

See also Enums.

See also 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.

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, Country country, Province province)

public IList<Province> Choices2CreateAddress(Country country) {
    if (country == null) { return new List<Province>; }
    var q = from p in Container.Instances<Province>()
            where p.Country.Id == countryOfResidence.Id
            select p
    return q.ToList();
}
```

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

How to allow selection of multiple choices

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

1. The parameter should be an `IEnumerable` of a domain object type and either:

    - that domain type is a bounded set

    - *or* a set of choices is provided via an action `Choices` method.

2. The parameter type is an `IEnumerable` of string or integer and a set of choices is provided via an action `Choices` method.

3. The parameter type is an `IEnumerable` of an enum

The first case is illustrated in the following example:

```
public void AddStandardComments(IEnumerable<string> comments) {
    foreach (string comment in comments) {
        Comment += comment + "\n";
    }
}

public string[] Choices0AddStandardComments() {
    return new[] {
        "Payment on delivery",
        "Leave parcel with neighbour",
        "Send SMS on delivery"
    };
}
```

How to specify auto-complete for a parameter

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

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

[PageSize(10)]
public IQueryable<Customer> AutoComplete0CreateNewOrder( [MinLength(3)] string
name) {
  return CustomerRepository.FindCustomerByName(name);
}
```

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

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

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

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

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

Use the StringLength, MultiLine, Mask or RegularExpression attributes.

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

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

How to validate parameter values

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

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

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

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

```
public string Validate<ActionName>(<parameter type>
parameterNameSameAsOnAction)
//OR
public string Validate<ParameterNumber><ActionName>(<parameter type>
anyOldName)
```

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

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

```
public class Customer {

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

  public string ValidatePlaceOrder(Product p) {
    if (p.IsOutOfStock()) {
      return "Product is out of stock";
    }
    return null;
  }

  //Parameter number 2 is used to match up to 'purchaseOrderNumber'
  public string Validate2PlaceOrder(string pon)
  {
    if (! pon.StartsWith("PO"))
    {
      return "Purchase Order Number must start with 'PO'";
    }
    return null;
  }
}
```

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

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

  public string ValidateSpecifyApplicability(DateTime fromDate, DateTime
toDate) {
    if (toDate.Date < fromDate.Date)   {
      return "From date cannot be before To date";
    }
    return null;
  }
}
```

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

How to work with date parameters

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

How to specify conditions for invoking an action

Disabling an action based on the state of the object

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

The syntax is:

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

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

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

  public string DisablePlaceOrder(Product p, int quantity)
  {
    if (isBlackListed())
    {
      return "Blacklisted customers cannot place orders";
    }
    return null;
  }
}
```

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

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

Disabling multiple actions on an object

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

```
  public string DisableActionDefault() {
    if (!Container.IsPersistent(this)) {
      return "Cannot invoke this action until the object has been saved";
    }
    return null;
  }
```

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

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

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

Use the `MemberOrder` attribute.

How to hide actions

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

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

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

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

```
public class Order
{
  public void ApplyDiscount(int percentage) {...}

  public bool HideApplyDiscount()
  {
    return isWholesaleOrder();
  }
  ...
}
```

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

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

```
  public bool HideActionDefault() {
    return !Container.IsPersistent(this);
  }
```

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

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

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

How to pass a message back to the user

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

```
public void InformUser(string message)

public void WarnUser(string message)
```

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.

How to create an action that operates on an object's collection

See How to create an action that operates on a selection from a collection.

How to make an action appear within an object collection

Local collection actions, which operate on a selection of objects within an object collection, are automatically rendered within that collection when the collection is opened.

Sometimes it may be desirable to render other object actions within the context of a collection. A common example of this is an action that adds a new member to a collection (and which therefore does not take existing collection members as a parameter). This may be achieved by marking up the action with the `MemberOrder` attribute and specifying the `Name` parameter to match the name of the collection:

```
[MemberOrder(Sequence = "1", Name = "Details")]
public void AddNewDetail(Product product, short quantity) { }

private ICollection<SalesOrderDetail> details = new List<SalesOrderDetail>();

public virtual ICollection<SalesOrderDetail> Details {
    get { return details; }
    set { details = value; }
}
```

How to create an action that will allow multiple rows of data to be entered

By marking up an action with the `MultLine` attribute, the action will not open as a dialog embedded within the existign view, but as a full-screen view and with the parameters laid out horizontally, instead of vertically.

This view will also allow the user to invoke the same action dialog multiple times, with a record kept of the paramater values for each successful submission. The effect is like a 'data grid' style of data entry - and is very useful where the user must repeatedly enter data of the same type.

By default, six rows will be rendered, but this may be overridden by specifying the NumberOfLines parameter on the attribute e.g.

```
[MultiLine(NumberOfLines=1)]
```

If all the rendered rows have been used, then submitting the last row (whether it is at the bottom of the grid or not) will cause a new input row to be rendered.

If the submission of any row is unsuccessful, either as a result of failing parameter validation, or a server error, then its fields remain editable and with an OK button.

It is important to understand that:

- Each row submission (by hitting the OK button at the end of that row) is an *independent transaction*, irrespective of when it is submitted.
- Row submission is asynchronous, so the user may immediately start entering or editing other unsubmitted row(s), and even submit them, before the first submission has been fully processed.
- When the user is done with entering rows, the Close button may be invoked to return to the previous context. `Close` does NOT submit any rows. So the user should take care to ensure that all intended rows are successfully submitted before closing the view. To help make this clear, a count of the number of rows successfully submitted is rendered next to the `Close` button.

Authentication

The Naked Objects Template solution is set up to provide a straightforward approach to user authentication, where the application requires it. It makes use of the Auth0 login service, which, in turn, allows you to log in using your Google, Facebook, or other recognised account. To use this capability in your Naked Objects application, you will need to create an account on Auth0. You can start with the free version, and upgrade only if you need the more advanced features or level of service. Follow the instructions on their website to create the account, then:

1. Create a `New Client` of type `Single Page Web Application`
2. Select `Angular 2+` as the technology for the web app.
3. Give your new Client a suitable name (`My App` is the default), and make a note of these settings, which you will need to copy into your application code.
    a. The *Domain*, which you will have chosen when you set up your account e.g. `FooBar.eu.auth0.com`
    b. The *Client ID* (a string of letters and numbers)
    c. The *Client Secret* (another, longer, string of letters and numbers)
4. Configure the *Callback URLs* to recognise the Urls from which your client will call the Auth0 service. If you are using the Naked Objects Template project unmodified then the default client Url is http://localhost:5001, but if you move this onto another port or another server (e.g. Azure) you will need to register the new Url here.
5. Under *Connections*, specify the forms of authentication your application will permit. For example under *Connections* > *Social* you might specify Google, Facebook, Amazon, and GitHub.
   To get started, the following settings are recommended:
   ```
   Use Auth0 instead of the IdP to do Single Sign On = false
   Advanced Settings > OAuth > JsonWebToken Signature Algorithm = HS256
                               OIDC Conformant = false
                               Grant Types > enable everything
   ```

Having now completed the set up of your Auth0 account, you will need to modify the project code to make use of this. You need to make changes to both the Client and Server projects.

Starting with the client project, go to the `config.json` file and set:

1. `"authenticate": true` client
2. `authDomain` to the *Domain* that you made a note of (above).
3. `authClientId` to the *Client ID* that you made a note of (above).

Now on the server project:

1. Add the `[Authorize]` attribute onto the `RestfulObjectsController` at class level.
2. Add the following code into the server's `appsettings.json`.

```
    "Auth0": {
        "Domain": "nakedobjects.eu.auth0.com",
        "Audience": "bB8ln8mh8hVb5kxnpTowceMb0t25Owl9"
    }
```

Having now enforced authentication at both client and server level you might now want to manage the specific aspects of the application that a particular user can see and/or use; we call this Authorization.

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 two 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.

- **Custom authorization**. In this approach you plug-in your own mechanisms for managing authorization, which may be self-contained or may delegate responsibility to an external service. Using this approach it is possible to control authorization at the level of individual object instances.

Attribute-based Authorization

In this approach, authorization rules are implemented by applying attributes to properties and/or actions inside the domain model classes. If any property or action does not have an authorization attribute specified then that property/action is assumed to be available to all users.

Consider the following example:

```
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 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 {...}
```

This then applies the authorization rules to all properties and/or actions within that class. The class-level attributes will over-ride any such attributes applied to individual properties or actions within that class, including properties/actions specified on sub-classes. Attempting to mix the two approaches is not recommended. On initialisation of the system, if such an over-ride is detected, a system warning will be raised.

Custom Authorization

Custom authorization is specified using one or more authorizer classes, each which implements *either* `ITypeAuthorizer<T>,` where `T` is a concrete type from your domain model, or `INamespaceAuthorizer` (to provide authorization logic for any types within a specified namespace).

Each authorizer class must be registered.

When the framework needs to determine authorization for a given object, it will search for the authorizer that provides the most specific match to that type object; if no such authorizer is found, it will use the default authorizer, which by implementing `ITypeAuthorizer<object>` will be able to work with domain objects of all types. Thus, you will typically only add a type authorizer for types (or groups of types within a namespace) that require their own specific approach to authorization.

The code below shows a skeletal implementation of the `FooAuthorizer`:

```
public class FooAuthorizer : ITypeAuthorizer<Foo> {
  public bool IsEditable(IPrincipal principal, Foo target, string memberName)
{...}
  public bool IsVisible(IPrincipal principal, Foo target, string memberName)
{...}
}
```

- `IsVisible` is called to determine whether a given object-member (a property or an action method) should be visible to the current user. The user-name may be derived from the `principal` parameter; the specific object instance is passed in as the `target` parameter (this is needed only if you are writing 'instance-based authorization' logic); the third parameter gives the `memberName`.

- `IsEditable` is called to determine whether a given object property may be edited by the user. `IsEditable` has no determined meaning for action members.

Within the IsVisible and IsEditable methods, you can either write custom logic, or delegate out to some external service.

There is a useful method in TypeUtils (within the NakedObjects.Helpers assembly) that facilitates type-safe testing of a property name, for example:

```
TypeUtils.IsPropertyMatch<Customer, dateTime>(target, memberName, x =>
x.DateOfBirth).
```

This is especially useful within an implementation of INamespaceAuthorizer as the method checks both the type of the target and the match for the memberName.

An implementation of ITypeAuthorizer<T> or INamespaceAuthorizer can have domain services and/or the IDomainObjectContainer injected into it.

Controlling visibility of columns in a table view

In a table view the authorizer will be called for each object instance being rendered in the table. If a given property is not visible for any of the instances (rows) in the table then that property will not be rendered as a column in the table. If the property is visible for any instance(s) being rendered then the column will be shown, but the value within it rendered only for those instances that authorize its visibility.

Adding further security measures

From a security perspective, Naked Objects offers the huge advantage that authorization is enforced both at the user interface and at the server API. If a given user is not authorized to invoke a particular action, then that action will not be rendered on the default user interface. Moreover, if a user deliberately sought to by-pass this limitation by interacting directly with the Restful API of the server, constructing the URL and any necessary JSON arguments by hand, then the attempt would fail. The same would apply to attempting to update a property of an object that the user was not authorised to modify.

One potential hole in this protection is in the persisting of a new object. In normal usage, the user invokes an action (typically called something like Create new…) that returns a transient object as a form for the user to complete and then hit the Save button. Unauthorised users will not be presented with the Create new… action. However, an unscrupulous user could mock up the necessary JSON representation of a transient objects, and invoke the persist resource, without needing access to the Create new… action at all. Alternatively, they could alter fields on a transient object that were not intended to be modified (or perhaps not even seen) via the user interface.

To prevent this, when the server creates the transient object it creates a list of all of the properties that are not intended to be modified by the user (whether because they are read-only or hidden), adds the identity of the user, and creates a hash of them. This hash is passed up to the client with the data as an ETag. When the user saves an object the client must provide the hash value, and this is validated against the user identity and object properties.

The hashing is done by a component called `InvariantStringHasher` that is registered in the DIConfig file, thus:

```
services.AddTransient<IStringHasher, InvariantStringHasher>();
```

You may replace the default component with your own implementation of `IStringHasher`.

***Note**: There is a rare circumstance in which a transient object has a derived field that is dependent upon a field that the user enters, which can result the hashes not matching. If you should encounter this issue, simply mark the derived property with* `[NotPersisted]`.

Alternatively, if you do not need this level of security (because your application is for internal users that can be trusted not to attempt to hack the system), you can replace `SimpleStringHasher` with `ConstantStringHasher` – the latter returns a constant string that takes no account of the field values and hence will always match up irrespective of any changes.

Auditing

Naked Objects provides a general purpose mechanism for recording user actions, either to support formal auditing, or just to provide a user-accessible mechanism for identify who did what. In order to use the audit capability, you need to create one or more 'audit services' and then register them.

Each time the user persists or updates an object, or invokes an action (on a service, or on a persisted object), the framework will look for the auditor that most precisely fits type of the object or service on which the action/update/persist has been invoked, and call the appropriate method, as defined on `IAuditor`:

```
void ActionInvoked(IPrincipal byPrincipal, string actionName, object onObject,
bool queryOnly, object[] withParameters);

void ActionInvoked(IPrincipal byPrincipal, string actionName, string
serviceName, bool queryOnly, object[] withParameters);

void ObjectUpdated(IPrincipal byPrincipal, object updatedObject);

void ObjectPersisted(IPrincipal byPrincipal, object persistedObject);
```

Note that the `queryOnly` parameter will be set to true if the action can be determined by the framework to be 'query only' i.e. any action that returns an `IQueryable<T>` and/or that is marked up with the `QueryOnly` attribute.

The implementations of `IAuditor` may manipulate the provided details of the user action and then optionally persist them as special ‘audit record’ domain objects. The persisted audit records may then be made available to suitably-authorised users via, say, an `AuditRepository` service.

Note that any implementation of `IAuditor` can have domain services and/or the `IDomainObjectContainer` injected into it.

Profiling

If you want to profile your application, you can hook into various events generated by the framework, and then write out to a console, file, or logging framework. The pattern is somewhat similar to the patterns for Authorization and Auditing. Start by writing your own implementation of `NakedObjects.Profile.IProfiler`, for example:

```
public class MyProfiler : IProfiler {

    public void Begin(IPrincipal principal, ProfileEvent profileEvent,
                      Type onType, string memberName) {
        Debug.WriteLine(profileEvent.ToString() + " Start:" + DateTime.Now);
    }

    public void End(IPrincipal principal, ProfileEvent profileEvent,
                    Type onType, string memberName) {
        Debug.WriteLine(profileEvent.ToString() + " End:" + DateTime.Now);
    }
}
```

`ProfileEvent` is an enum that defines the set of Naked Objects events that can be profiled:

```
public enum ProfileEvent {
    ActionInvocation,
    PropertySet,
    Created,
    Deleted,
    Deleting,
    Loaded,
    Loading,
    Persisted,
    Persisting,
    Updated,
    Updating,
}
```

Next you need to specify which of these events you wish to profile as a `set`, for example:

```
new HashSet<ProfileEvent>() { ProfileEvent.ActionInvocation,
ProfileEvent.Updating, ProfileEvent.Updated };
```

Note that in addition to `ActionInvocation`, there are events that correspond to the Life Cycle methods for a domain object. Note: *you do not have to have any of those methods explicitly on your domain object - the events are generated whether or not you need to add behaviour into them via explicit methods*.

When profiling a Naked Objects application you would not normally use the `PropertySet` event, as this profiles only the time taken on domain code behaviour associated with an individual property set - not the total time to update the object. If you wish to profile the time taken from when a use hits Save on an object-edit view, then you can monitor from the start of `Updating` to the end of `Updated`.

Finally, you must configure the profiling in `DIConfig`.

Other how-to's

How to get 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.

How to create 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`.

This service can be injected into any domain object with a suitable property of type `NakedObjects.Snapshot.IXmlSnapshotService`, an interface which is defined in `NakedObjects.Helpers.dll` (installed into your model project as part of the `NakedObjects.ProgrammingModel` package). You can then call the `GenerateSnapshot` method on this service, passing in the domain object of interest. Having generated the snapshot, *but before reading the XML itself* (via its `Xml` property), you can also specify associated objects that should be in-lined within the snapshot - navigating as far down the graph of associated objects as you wish. All these ideas are illustrated in the code below, but we recommend that the best way to learn how this works is simply to experiment:

```
public class Product {

    ...

    //Injected service
    public ProductRepository Snapshotter {set; protected get}

    public void Archive() {
      IXmlSnapshot = Snapshotter.GenerateSnapshot(this);
      snap.Include("Manufacturer"); //In-lines an XML representation of the
object in the Product's Manufacturer property
      snap.Include("Manufacturer/Address"); //In-lines an XML representation of
the object in the Manufacturer's Address property
      string xml = snap.Xml;
      ...
    }
}
```

`IXmlSnaphot` also provides methods to return the XML schema definition (`Xsd`), and to transform the generated XML using a standard XSL Transform passed in as a string (`TransformedXml`).

How to run Naked Objects as a .exe

In the Naked Objects source code, root directory, there is a solution file named `Batch.sln` which, when opened, show example code for running Naked Objects as a .exe, to run periodic batch processes.

Note that:

- The application is configured, as usual in `NakedObjectsRunSettings`.
- `DIConfig` includes this additional line:
  `services.AddScoped<IBatchRunner, BatchRunner>();`
- `BatchStartPoint` may be replaced by your own class, implementing `IBatchStartPoint`.

You may choose to leave this `.exe` running permanently on your server, or to launch it at regular intervals via the Windows Task Manager.

If your batch implementation can be broken down in a series of separate tasks, then, rather than performing them sequentially, you should consider using the Naked Objects Async capability...

How to run multiple threads asynchronously

Naked Objects provides a convenient mechanism for running multiple Naked Objects threads in parallel. You will need to inject a `NakedObjects.Async.IAsynchService` (defined within the `NakedObjects.ProgrammingModel` package) into your code:

```
public IAsyncService AsyncService { set; protected get; }
```

(The Naked Objects framework provides an implementation of `IAsyncService` `NakedObjects.Async.AsyncService`. You will need to register this service, or another implementation if you wish to write your own).

You can call the `RunAsync` method on this service, passing in the action to be run as a lambda. This method returns a `System.Threading.Tasks.Task`; by collecting these tasks into an array you can then instruct the system to wait until all such tasks are completed, using the standard .NET code of `Task.WaitAll(tasks)`, as illustrated in the following example code:

```
public ProcessRepository ProcessRepository { set; protected get; }
public AsyncService AsyncService { set; protected get; }

public void RunAllProcessesDueBy(DateTime dateTime) {
    IList<ProcessDefinition> due =

ProcessRepository.GetProcessesWithNextRunDueBy(dateTime);
    var tasks = due.Select(pd => AsyncService.RunAsync((domainObjectContainer)
=>
                            FindAndRunProcess(pd.Id))).ToArray();
    Task.WaitAll(tasks);
}

private void FindAndRunProcess(IDomainObjectContainer container, int processId)
{
    var repository = container.GetService<ProcessRepository>();
    var proc = repository.GetProcess(processId);
    proc.Run();
}
```

In the example above, the `ProcessRepository` and `ProcessDefinition` are straightforward domain classes. The first line just obtains a set of objects that contain details of processes due to be run by the specified `DateTime`, each with a `Run()` method. The second line invokes a method on to find and run each of those due processes, asynchronously. The third line waits until all the asynchronous processes are completed.

Each of the asynchronously-run processes is fully independent. You should therefore *not pass domain objects into or between these processes*, but you may pass .NET primitives, including object Ids - to be retrieved independently inside the new thread. This is why, in the above example, `FindAndRunProcess` is passed in the Id of the `ProcessDefinition`, not the domain object itself, nor the `ProcessRepository`. The method RunAsync can pass in a properly set-up instance of `IDomainObjectContainer` into the method to be run, and this can then be used to obtain domain service(s) and, thence, domain instances. **Note**: The Naked Objects `AsyncService` is designed only to be used within a standalone executable run project (as illustrated above). It will not work within an HttpContext.

Patterns and practices

Working with Entity Framework

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

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

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

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

Overriding the default database schema generation

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

```
namespace DataAccessFluent {
    public class DestinationConfiguration :
EntityTypeConfiguration<Destination>
    {
        public DestinationConfiguration() {
            Property(d => d.Name).IsRequired();
            Property(d => d.Description).HasMaxLength(500);
            Property(d => d.Photo).HasColumnType("image");
        }
    }

    public class LodgingConfiguration : EntityTypeConfiguration<Lodging> {
        public LodgingConfiguration() {
            Property(l => l.Name).IsRequired().HasMaxLength(200);
            Property(l => l.Owner).IsUnicode(false);
            Property(l => l.MilesFromNearestAirport).HasPrecision(8, 1);
        }
    }
    //...
    public class BreakAwayContextFluent : DbContext {
        public BreakAwayContextFluent(string name) : base(name) { }
        public BreakAwayContextFluent() { }
        public DbSet<Destination> Destinations { get; set; }
        public DbSet<Lodging> Lodgings { get; set; }
        //...
        protected override void OnModelCreating(DbModelBuilder modelBuilder) {
            modelBuilder.Configurations.Add(new DestinationConfiguration());
            modelBuilder.Configurations.Add(new LodgingConfiguration());
            // ...
        }
    }
}
```

Using data fixtures with Code First

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

```
namespace DataAccess {
    public class DropCreateBreakAwayWithSeedData :
                              DropCreateDatabaseAlways<BreakAwayContext> {
        protected override void Seed(BreakAwayContext context) {
            context.Destinations.Add(new Model.Destination {
                                                     Name = "Great Barrier
Reef" });
            context.Destinations.Add(new Model.Destination { Name = "Grand
Canyon" });
            //...
        }
    }
}
```

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

How to 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. The following code shows an example of eager loading within a query method - ensuring that the object referenced within the `Customer`'s `Pet` property is loaded at the same time as the `Customer`:

```
using system.data.entity;

var petOwners = Container.Instances<Person>().Where(x => x.HasPet).Include(x =>
x.Pet);
```

How to implement complex types

If a domain object has a property that is a Complex Type, then this will be rendered the same way as a regular reference property (i.e. an association), complete with the title of the complex type object. However, the url in the link just points to the parent object (i.e. the object you are currently viewing). There is currently no way to view or edit the properties of the complex type object directly.

If you need to use a Complex Type and you need to view and/or edit the properties of that complex type (other than just viewing the title) then you will need to add one or more dedicated actions, as illustrated with the following example.

```
    public class Customer {
        //This property is a complex type, see below
        [Disabled]
        public Address HomeAddress { get; set; }

        public void EditAddress(string line1, string line2) {
            HomeAddress.Line1 = line1;
            HomeAddress.Line2 = line2;
        }

        public string Default0EditAddress() {
            return HomeAddress.Line1;
        }

        public string Default1EditAddress() {
            return HomeAddress.Line2;
        }
    }

    [Inline] //Complex Type
    public class Address {
        //Title (would probably truncate each line)
        public string ToString() {
            return Line1 + Line2;
        }
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        //etc
    }
```

Note: When working Code First it is necessary to annotate the complex type class with the `ComplexType` attribute, *even if you have specified a complex type by means of the Code First Fluent API*. This is because Naked Objects needs to be able to detect that it is a complex type independently of Entity Framework.

How to work with multiple databases

Naked Objects can cope with multiple databases. You will need to add the `DbContext` for each database in the `EntityObjectStoreConfiguration`. See Creating a Model project from scratch.

How to work with multiple database contexts

Naked Objects provides very good support for working with multiple contexts as shown below:

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new ModelCContext());
    config.UsingCodeFirstContext(() => new ModelDContext());
    return config;
}
```

These various contexts may correspond to separate databases, or they may point to the same shared database(s).

At run-time, when an object is retrieved (whether explicitly via `Container.Instances<T>`, or just by navigation) Naked Objects transparently identifies which of the contexts the type resides in and instructs Entity Framework to retrieve the object from that context. This is a very powerful capability, and enables things like polymorphic association.

However, this pattern does carry an overhead, that grows with the number of contexts you have. This is because - due to a severe limitation in the design of Entity Framework - the only way to find out if a context knows about a given domain type is to ask for that type, and catch the exception thrown if that type is not known. Exceptions are relatively expensive in processing terms, so raising lots of them - from polling multiple contexts - can be slow. Naked Objects does cache the mapping of types to contexts, but this cache is generated progressively, as each type is encountered.

This overhead can be eliminated by explicitly associating types with their context, using the syntax below:

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new
       ModelAContext()).AssociateTypes(ModelATypes);
    config.UsingCodeFirstContext(() => new
             ModelBContext()).AssociateTypes(ModelCContext.ModelBTypes);
    …
    return config;
}

private static Type[] ModelATypes() {
    return new Type[] {typeof(Payment), typeof(Invoice), ... };
}
…
```

The `AssociateTypes` method (on `ContextInstaller`, which is returned by `Using...Context`) takes as a parameter any method that returns an array of types. Each of these specified associations will be cached on the session. The methods defining the array of types could be local - as shown for `ModelATypes()` above - or as an external static method, for example, on the appropriate code first `DbContext` - as shown for `ModelCTypes` above.

Although it is sensible to prime the cache with all the types that you will be using, there is no requirement to do so. If the framework comes across a type for which it does not know the context, it will poll the contexts to find it, and then cache the association - albeit with the overhead mentioned previously.

As a further refinement to this performance optimisation, there is a method `SpecifyTypesNotAssociatedWithAnyContext` on the `EntityObjectStoreConfiguration`. This method only adds value where you have a domain class (typically an abstract class, though not necessarily) that is not itself an entity - and therefore unknown to Entity Framework - but from which one or more entity types inherit.

A simple example of this exists within our AdventureWorks sample project, where many domain classes inherit from `AWDomainObject` - which provides a small amount of functionality common to many classes. In ordinary operation, Naked Objects will work up the inheritance hierarchy, until it finds a class that is unknown to any DbContext, and then work with the next level down. This incurs the overhead of a thrown and caught exception mentioned earlier. To improve efficiency, the AdventureWorks sample project now contains this code:

```
public static EntityObjectStoreConfiguration EntityObjectStoreConfig() {
    var config = new EntityObjectStoreConfiguration();
    config.UsingCodeFirstContext(() => new MyContext())
      .AssociateTypes(AllPersistedTypesInMainModel)
      .SpecifyTypesNotAssociatedWithAnyContext(() => new[] {typeof
(AWDomainObject)});
    return config;
}
```

If there was more than just the class `AWDomainObject` to which this applied, then the method may be called with a delegate, in a manner similar to `AssociateTypes` e.g.:

```
installer.SpecifyTypesNotAssociatedWithAnyContext(TypesToIgnore);
```

The `EntityObjectStoreConfiguration` has a `RequireExplicitAssociationOfTypes`, which is set to false by default. If set to true when initialised, then the framework will throw an uncaught exception if the persistor is asked for any domain type that has not either been associated with a specific DbContext (using `AssociateTypes`) or with no context (using `SpecifyTypesNotAssociatedWithAnyContext`).

Writing safe LINQ queries

Don't use the equality operator on objects; test for equality on the value properties

Don't write:

```
public IQueryable<Product> FindProducts(ProductCategory category) {
  return Container.Instances<Product>().Where(x => x.Category == category);
```

Write:

```
public IQueryable<Product> FindProducts(ProductCategory category) {
  return Container.Instances<Product>().Where(x => x.Category.Id ==
category.Id);
```

Don't call any method on a domain object within a query; refer only to properties

Don't write:

```
public IQueryable<Product> ListDiscontinuedProducts() {
  return Container.Instances<Product>().Where(x => x.IsDiscontinued());
```

Write:

```
public IQueryable<Product> ListDiscontinuedProducts() {
  return Container.Instances<Product>().Where(x => x.Status == "Discontinued");
```

Note, though that you can call methods on System classes e.g. Trim().ToUpper() on string.

Don't navigate references on any objects passed into the query; pass in any such required indirect references as variables in their own right

Don't write:

```
public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {
  return Instances<Order>().Where(x => x.SentTo.Id != cust.BillingAddress.Id);
```

Write:

```
public IQueryable<Order> FindOrdersNotSentToBillingAddress(Customer cust) {
  Address billingId = cust.BillingAddress.Id;
  return Instances<Order>().Where(x => x.SentTo.Id != billingId);
```

When doing a join, don't try to use Container.Instances<T>() more than once inside a query; define a separate IQueryable<T> outside the query

Don't write:

```
var q = from p in Container.Instances<Product>()
        from c in Container.Instances<Customer>()
        where ...
```

Write:

```
var customers = Container.Instances<Customer>();
var q = from p in Container.Instances<Product>()
        from c in customers
        where …
```

Or, for greater clarity:

```
var customers = Container.Instances<Customer>();
var products = Container.Instances<Product>();

var q = from p in products
        from c in customers
        where …
```

Polymorphic Associations

Entity Framework requires that any association is defined by a type that a Class (whether concrete or abstract). It does not natively support the concept of an association that is defined by an Interface.

To overcome this limitation of Entity Framework, *Naked Objects* provides two patterns for achieving the goal: the 'result interface association' and the 'polymorphic association'.

Result interface association

This is the simpler case where a property is defined by an interface but there is expected to be just a single implementation of that interface. In such cases the Interface is not defining a role for multiple objects to play - it is simply defining a reduced view of an object that is returned as the result of a query, say, perhaps in order to hide other aspects of that object's implementation. The 'role interface association' pattern as described above would still work here, but it is overkill. The specific 'result interface association' pattern is simpler to code, and potentially faster in execution. (It also means that the relationship may be implemented at database level as a foreign key, which the role interface association does not permit.)

The example below shows how to code the pattern.

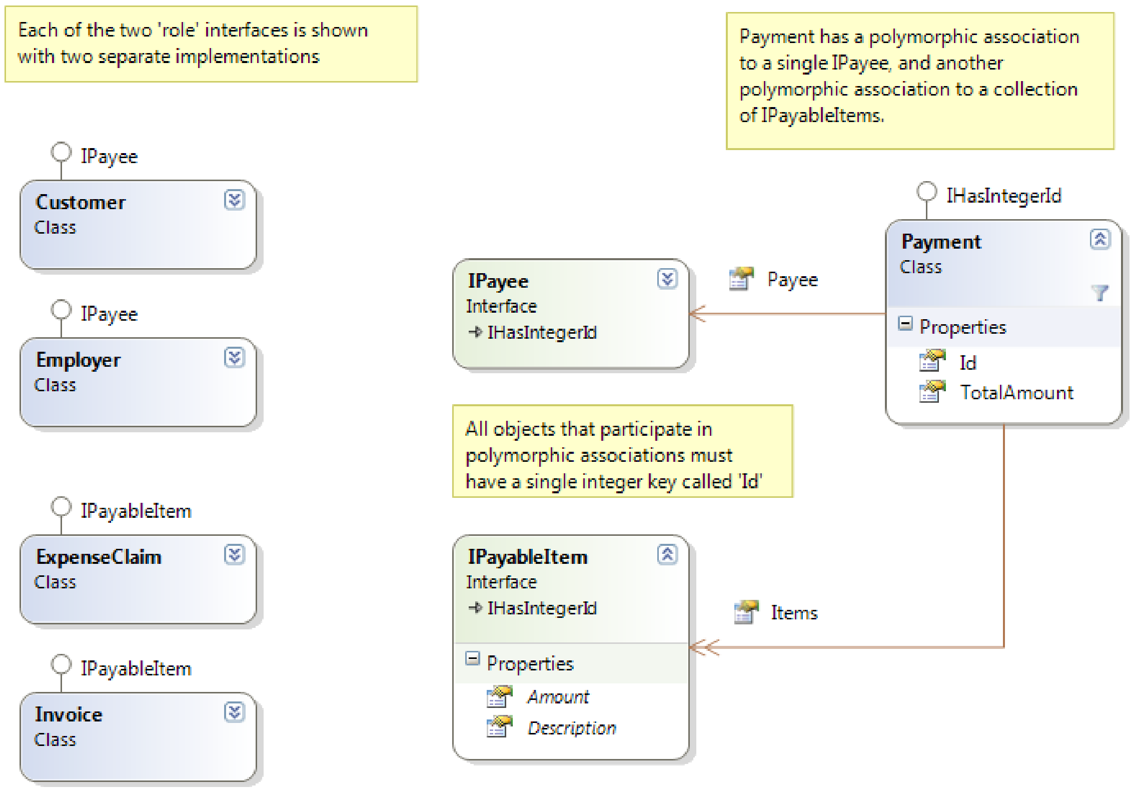
Use the Result Interface Association snippet (shortcut `resultia`) to help create this code.

```
public class Order {

   [Hidden()]
  public virtual int CustomerId {get; set;}

  private ICustomer _Customer;

 [NotPersisted()]
  public ICustomer Customer {
    get {
      if (_Customer == null && CustomerId > 0) {
        _Customer = CustomerRepository.FindById(CustomerId);
      }
      return _Customer;
    }
    set {
      _Customer = value;
      if (value == null) {
        CustomerId = 0;
      }
      else {
        CustomerId = value.Id;
      }
    }
}
```

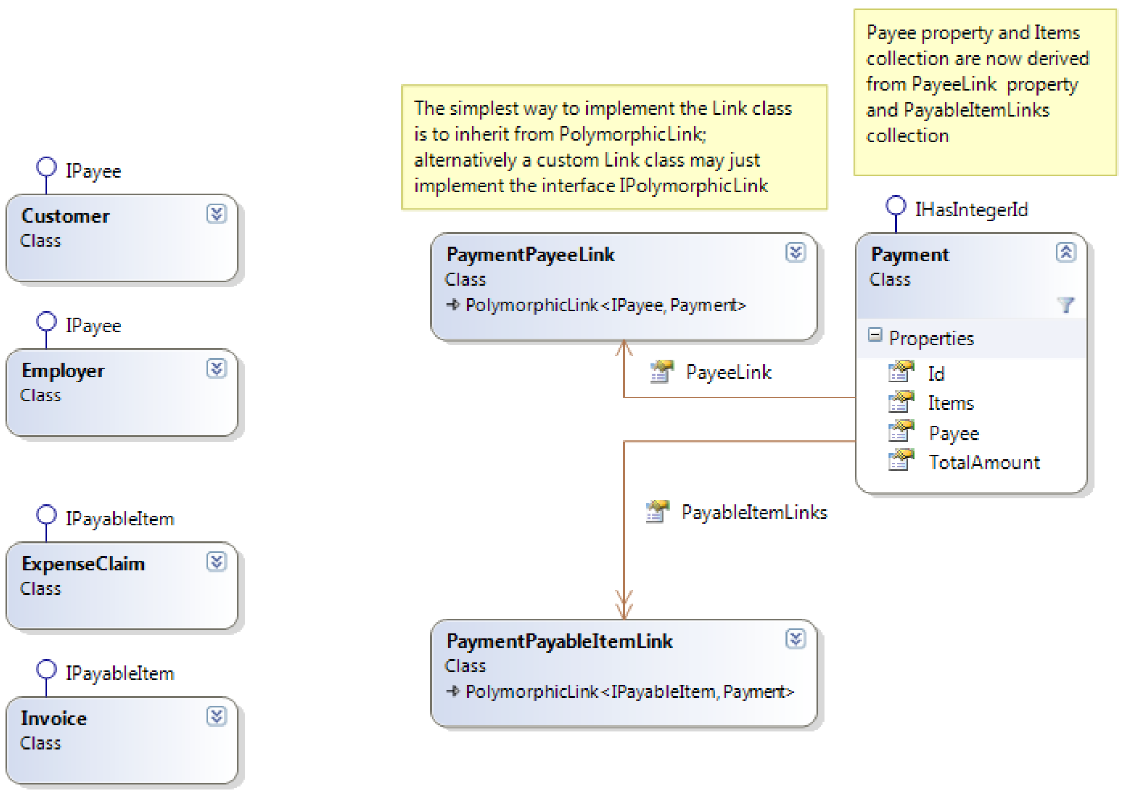
Polymorphic Association

Polymorphic Associations (PAs) - associations between objects where the type is defined by an ‘role’ interface rather than by a concrete or abstract class.

The following example is based on a Payment object that has two PAs. The first is a single association to a Payee, defined by the role interface IPayee. In the example there are two separate implementations of IPayee: Customer and Employer. In practice there may be many more. The second PA is a multiple association, to a collection of type IPayableItem. In this example there are two implementations of the role IPayableItem: ExpenseClaim and Invoice. The intent of the domain model is represented in this diagram:


Each of the two 'role' interfaces is shown with two separate implementations
Payment has a polymorphic association to a single IPayee, and another polymorphic association to a collection of IPayableItems.
All objects that participate in polymorphic associations must have a single integer key called 'Id'
IPayee
Customer
Class
IPayee
Employer
Class
IPayableItem
ExpenseClaim
Class
IPayableItem
Invoice
Class
IPayee
Interface
IHasIntegerId
Payee
IHasIntegerId
Payment
Class
Properties
Id
TotalAmount
IPayableItem
Interface
IHasIntegerId
Properties
Amount
Description
Items

Next we turn to the implementation of the PA - initially from a domain modelling perspective, as shown below:


Payee property and Items collection are now derived from PayeeLink property and PayableItemLinks collection
The simplest way to implement the Link class is to inherit from PolymorphicLink; alternatively a custom Link class may just implement the interface IPolymorphicLink
IPayee
Customer
Class
IPayee
Employer
Class
IPayableItem
ExpenseClaim
Class
IPayableItem
Invoice
Class
PaymentPayeeLink
Class
PolymorphicLink<IPayee, Payment>
PayeeLink
PayableItemLinks
PaymentPayableItemLink
Class
PolymorphicLink<IPayableItem, Payment>
IHasIntegerId
Payment
Class
Properties
Id
Items
Payee
TotalAmount

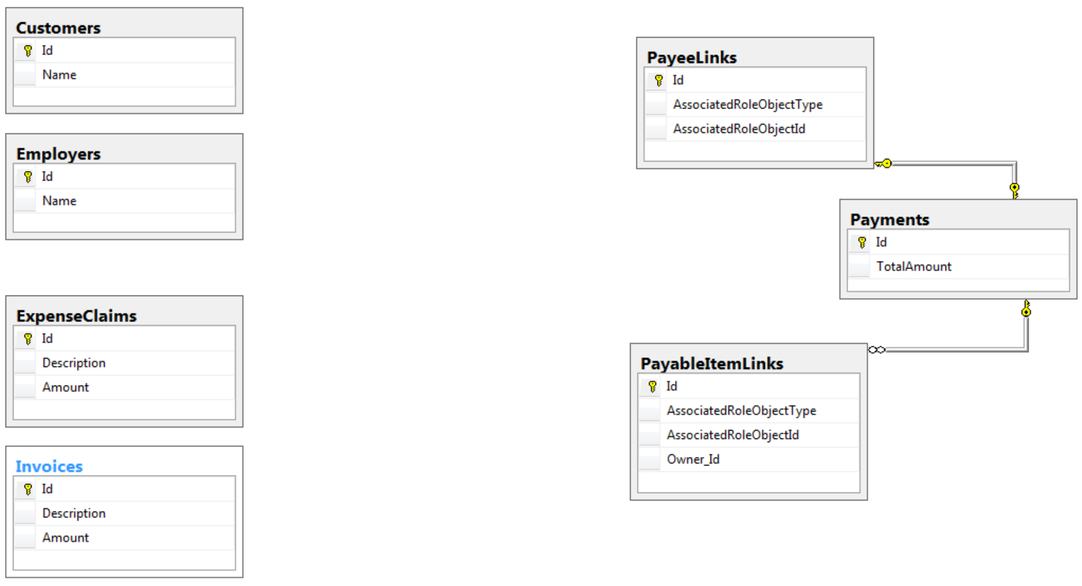
Note the addition of two new domain classes: PaymentPayeeLink and PaymentPayableItemLink. Both of these inherit from a generic helper class - PolymorphicLink<TRole, TOwner> - which is defined in the NakedObjects.Helpers assembly, installed by the NakedObjects.ProgrammingModel package.

For this particular example, the two new classes do not add any properties or methods to the generic helper class, they are simply defined thus:

```
public class PaymentPayeeLink : PolymorphicLink<IPayee, Payment> { }

public class PaymentPayableItemLink : PolymorphicLink<IPayableItem, Payment>
```

However, these Link classes could, in principle, add custom properties and/or methods. A particular use of this might be to de-normalise some properties (such as a Title property or other summary information) from the associated object.

Working CodeFirst, each of these Link domain objects will result in an equivalent table in the database. This is shown in the database diagram below (table names have been shortened for clarity):



Note that each of the Link tables has a foreign key reference back to the 'owner' (in this case a Payment). In the case of PayeeLinks, the foreign key is also the Id - because it is a single association. Each Link table has two columns that, in combination, specify the associated object: AssociatedRoleObjectType and AssociatedRoleObjectId. The latter is always an integer (one constraint of this pattern is that all objects that participate in the PA must have a single integer Id property called 'Id' in the code (the naming convention in the database is not mandated). The AssociatedRoleObjectType is a string, and may either take the form of the fully-qualified type name (e.g. 'MyApp.Customer') or a standard abbreviation (e.g. 'CUS') - this is configurable and may even be mixed. These two items permit Naked Objects to retrieve the - corresponding object - using methods provided by the generic PolymorphicLink class (which, in turn, delegates to an injected service called PolymorphicNavigator, which is also included in the NakedObjectsProgramming model).

Using the code snippets

Almost all the domain code needed to implement this pattern is generated by the use of two code snippets (see Using the Naked Objects Snippets):

- `polyprop` for a polymorphic property such as the payee in the example above.
- `polycoll` for a polymorphic collection such as the payable items in the example above.

You will need to register the NakedObjects.PolymorphicNavigator as a service.

The code below shows code from the example that was generated entirely by the `polyprop` snippet:

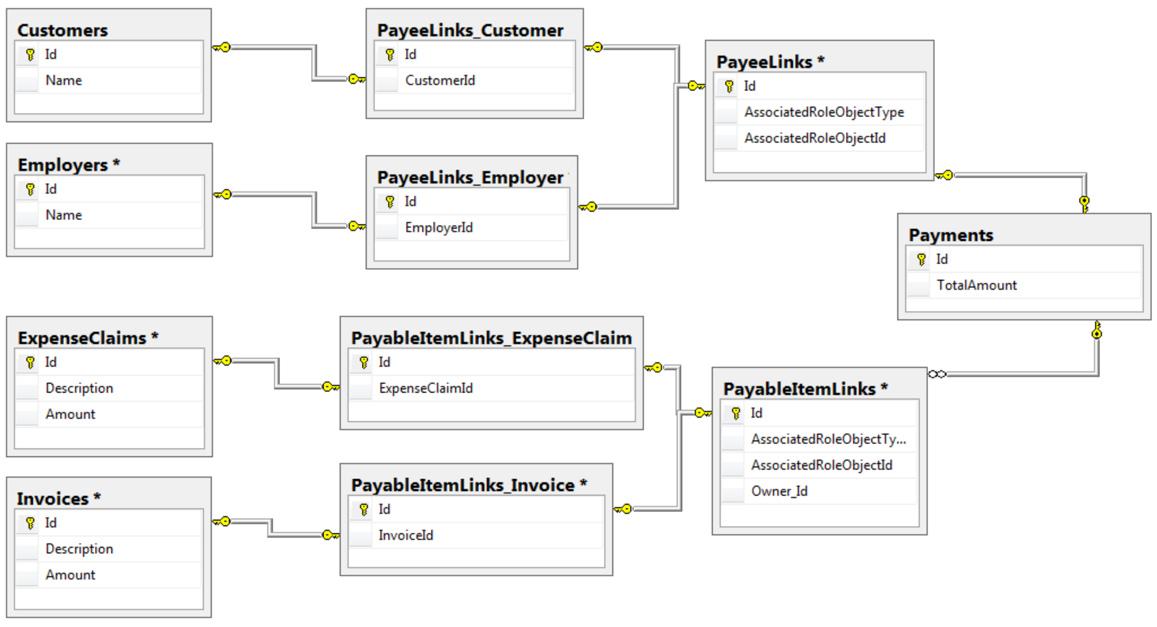
```
#region Payee Property of type IPayee ('role' interface)

[Disabled]
public virtual Payment_Payee_Link PayeeLink { get; set; }

private IPayee _Payee;

[NotPersisted]
public IPayee Payee {
  get {
    return PolymorphicNavigator.RoleObjectFromLink(ref _Payee, PayeeLink,
this);
  }
  Set {
    _Payee = value;
    PayeeLink = PolymorphicNavigator.UpdateAddOrDeleteLink(_Payee, PayeeLink,
this);
  }
}

public void Persisting() {
  PayeeLink = PolymorphicNavigator.NewTransientLink
                  <Payment_Payee_Link, IPayee, PolymorphicPayment>(_Payee,
this);
}
#endregion
```

The example code below was generated using the `polycoll` snippet:

```
#region PayableItems Collection of type IPayableItem

private ICollection<Payment_PayableItem_Link> _PayableItem =
    new List<Payment_PayableItem_Link>();

[NakedObjectsIgnore]
public virtual ICollection<Payment_PayableItem_Link> PayableItemLinks {
  get  {
    return _PayableItem;
  }
  set {
    _PayableItem = value;
  }
}

public void AddToPayableItems(IPayableItem value) {
  PolymorphicNavigator.AddLink
            <Payment_PayableItem_Link, IPayableItem, PolymorphicPayment>(value,
this);
}

public void RemoveFromPayableItems(IPayableItem value) {
  PolymorphicNavigator.RemoveLink
            <Payment_PayableItem_Link, IPayableItem, PolymorphicPayment>(value,
this);
}

[NotPersisted]
public ICollection<IPayableItem> PayableItems {
  get {
    return PayableItemLinks.Select(x => x.AssociatedRoleObject).ToList();
  }
}
#endregion
```

Adding optional (database) referential integrity to a polymorphic association

As may readily be observed from the diagram above, at this stage the database schema does not have referential integrity across the PA; and navigating for reporting purposes, while possible, will not be easy. If this is an issue for your application, then it may optionally be addressed added by manually adding further tables as shown in the example below:



Notice that for each of the polymorphic link tables (on the right hand side) there is now an added table *for each of the implementations of that role.* Thus PayeeLinks has associated PayeeLinks_Customer and PayeeLinks_Employer tables. Each of these ‘second half’ tables (on the left-hand side of the diagram) has an FK both to a row in the corresponding Link table, and to its ‘role’ object. These ‘second-half’ tables will need to be created manually, and maintained by custom database triggers: they are invisible to Entity Framework and the domain code.

The cluster pattern

The Cluster Pattern is a specific high-level pattern for breaking up a large domain object model – following a very strict set of rules. Each cluster provides a distinct, re-usable, piece of business functionality. The term 'cluster' has been used only to distinguish this pattern from the more general idea of re-usable 'modules' or 'components' which, although they might have some features in common with the cluster pattern, typically do not enforce such strict rules.

The Cluster Pattern does not depend upon the Naked Objects Framework (NOF). However, the NOF makes it easier to implement the pattern, and makes its value more explicit.

Hard Rules

1. Each Cluster is defined by separate Api and Impl projects - the latter referencing the former. The Api defines only the programmatic interface to the Cluster (as may used by other Clusters) - it does not define the user view of the cluster. In general the Api exposes the minimum possible of the implementation.

2. A cluster depends on other clusters only where this clearly makes sense from a business perspective. For example, the Emails cluster depends upon the Documents cluster as the created emails may be stored as documents.

3. A Cluster may only ever reference the Api of another Cluster; it may never reference another Impl project. (It is recommended that this be enforced by build script rules on your build server).

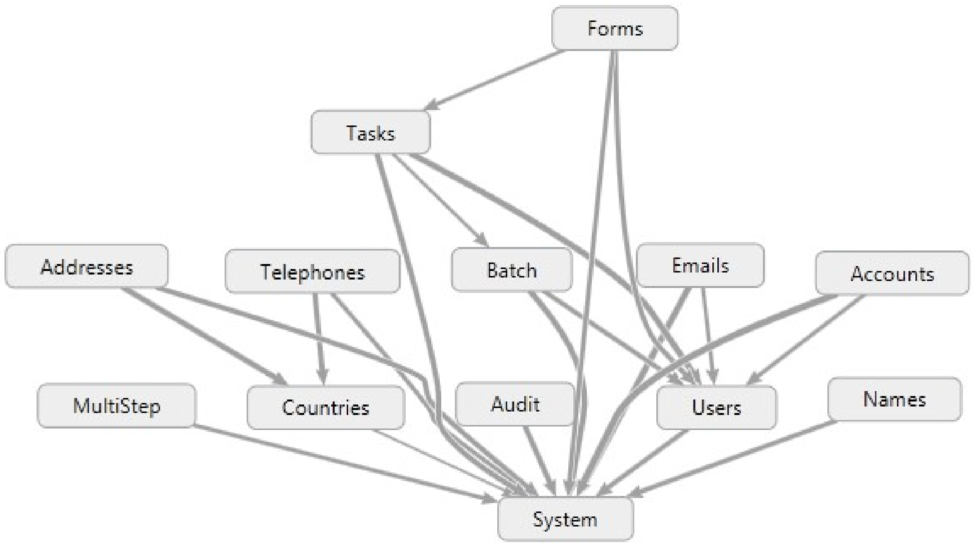
4. A Cluster Api will typically consist of interfaces. An Api may also contain Enums, Constants, and (less commonly) static classes and methods. Members on interfaces should use only other interface types, Enums, or .NET value types. An Api may NOT contain any persistable classes, whether abstract or concrete. They may contain ViewModels or other non-persistent classes, though this is not encouraged as it can lead to confusion. It follows from the above that:

- Classes in a Cluster Impl may not inherit from classes in other clusters: this is a deliberate constraint.
- Any associations between objects in different clusters must be defined by interfaces, and must therefore follow a Polymorphic Association pattern.

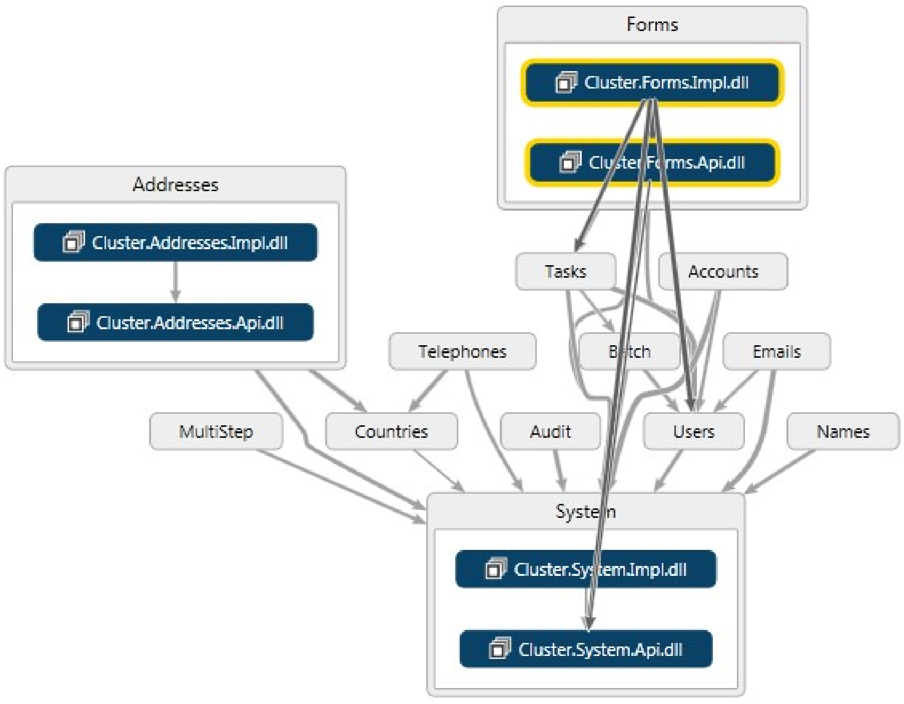
5. The interfaces in a Cluster Api are explicitly labelled as one of three types:

- A **service** interface defines a service for which there is an implementation in the Impl. Other clusters may require an implementation of this service interface injected into them.
- A **result** interface is a restricted view of a domain type that is defined within the cluster’s Impl and may be created and/or retrieved by means of service interface methods. Good practice says that result interfaces should hide data behind higher-value behaviours where possible i.e. provide methods in preference to properties; any properties that *are* exposed should be read-only except in rare cases – in which case the rationale should be documented with comments.
- A **role** interface is intended to be implemented by objects in *other* clusters in order that those objects can take advantage of behaviour implemented in the cluster. Thus, those role interfaces may form input-parameters for methods on the service interfaces defined in the API. Additionally, the cluster Impl may define ContributedActions for that role interface.

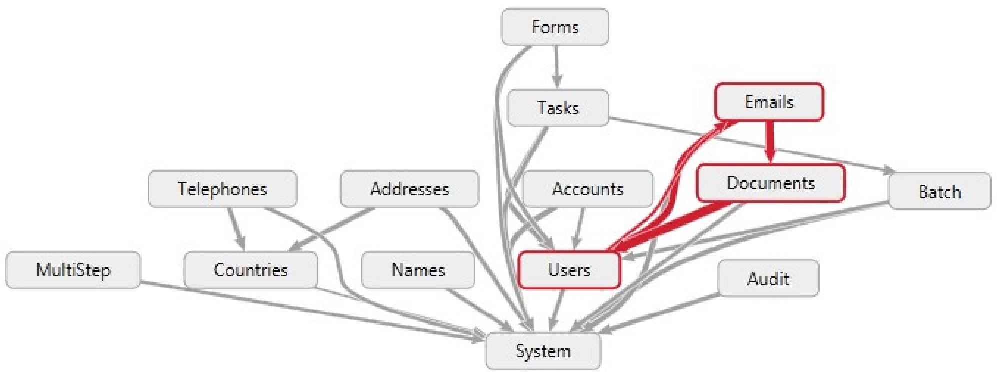
Clusters should form a natural hierarchy of depenency: if the Api and Impl were treated as a single entity there should be no cicular dependencies between them. This is a subtle point, not the same thing as saying that there should be no circular dependencies between projects (which would not compile anyway). This is illustrated in the following diagrams. First, an example of a cluster hierarchy:



Notice that each cluster depends only on cluster below it. Expanding the view of three of the clusters, we can see that each one has an Api and Impl project; that the Impl depends on the Api, and that the Api *and/or* the Impl projects may depend on the Api's in other clusters, but never on other impls:



Finally, here is a diagram of an earlier iteration of the same cluster project:



The diagramming tool (Visual Studio Ultimate in this case) has detected a circularity between clusters, shown in red. This is not strictly a circular reference between projects (as that would not compile). It occurs, in this example, because the Users.Impl project referenced Emails.Api, and Emails.Impl referenced Users.Api. in addition to the (intended) dependencies of Emails on Documents, and Documents on Users. The problem with this sort of circularity is that it means that Emails, Documents and Users cluster would always have to be used together and effectively form a 'mega-cluster'. This error was easy to correct, however.

Optional Rules

The following represent good practices which are followed in this code base, but are not strictly definitional to the cluster pattern:

6. Each cluster defines its own `DbContext` and mappings.

7. Each cluster manages its own authorization via a standard pattern. The roles used by this authorizer are defined as constant strings on the cluster API.

8. Each cluster has its own test project.

9. All projects in this library are set to All Warnings As Errors - so that they are warning free.

10. Clusters may be deployed as NuGet Packages, one each for the Api and Impl respectively, using SemVer rules for versioning.

Building the framework from source

The source code for the Naked Objects Framework is hosted on GitHub here: https://github.com/NakedObjectsGroup/NakedObjectsFramework

To build the framework locally from source, open a command window on the directory containing the framework source (e.g. C:\NakedObjectsFramework) and execute these four commands in sequence:

```
dotnet build ProgrammingModel.sln

dotnet pack "Programming Model\NakedObjects.ProgrammingModel.Package\
NakedObjects.ProgrammingModel.Package.csproj"  --include-symbols --include-
source

dotnet build Server.sln

dotnet pack Server\NakedObjects.Server.Package\
NakedObjects.Server.Package.csproj --include-symbols --include-source
```

The `NakedObjects.ProgrammingModel` and `NakedObjects.Server` NuGet packages will be found in the `bin` directories of the `…package.csproj` from which they were built.

Troubleshooting

Logging

If you are hitting run-time errors, and the source of the error is not clear from any exception being thrown, then a recommending step is to take advantage of logging.

Naked Objects adopts the standard .NET Core approach to logging, allowing different logging providers to be configured.

For an example of how to configure logging, look at the `Template.Server` project, which is configured to use the `Log4Net` logging framework.

The Log4Net framework (you will need to add the NuGet package `Microsoft.Extensions.Logging.Log4Net.AspNetCore`) is configured in the file `log4net.config`, and the use of this framework is specified in the `Startup.cs` file here:

```
loggerFactory.AddLog4Net();
```

This configuration records logging messages into the file `nakedobjects_log.txt` within the solution.

Logging from within the domain code

In addition to recording logging events from with the Naked Objects Framework code, it is also possible to log events from within domain code. To do this, add *either or both* of the following lines of code into a domain class:

```
public ILogger<T> Logger { set; private get; }
public ILoggerFactory LoggerFactory { set; private get; }
```

Naked Objects will then automatically inject an implementation of `ILogger` for the specified domain type T (e.g. `Customer`), and/or an implementation of `ILoggerFactory`, from which other types of logger may be accessed. Note that those two types, `ILogger<T>` and `ILoggerFactory` are both defined in `Microsoft.Extensions.Logging`, and represent an abstraction of specific logging frameworks.

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.

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.

Class not public

This error was caused by the domain class `FooBar` not being `public`.

The best way to avoid this error occurring is to create new domain classes using the Domain Object item template, which automatically creates a public class.

Errors thrown when running an application

This section contains guidance on error messages that might appear while you are using a Naked Objects application.

A property is not virtual/overrideable

This error has arisen because the class `FooBar` contains:

- One or more properties that have not been made `virtual`. This is a requirement in order to allow the Entity Framework to create a proxy for the object. The best way to avoid this error occurring is to create all new properties using the `propv` code snippet.

- A collection that is not of type `ICollection`. See Collection properties. The best way to avoid this error occurring is to create new collections using the `coll` code snippet.

Invalid column name

This error is quite common when prototyping in Code First mode with Entity Framework - where a new property (in this case called `Surname`) has been added to a class since the database was created. The solution is either to delete the database and run again (which will re-create the database), or to manually add a new column to the appropriate database table.

Invalid object name

As indicated by the fact that this is a `SqlException`, this error arises where a domain object class does not have a corresponding table in the database. The remedy is the same as for Invalid Column Name (above).

Collection not initialised

This error has arisen because a collection was not initialised. See Collection properties. The best way to avoid this error occurring is to create new collections using the `coll` code snippet.

Database is generated, but certain (or all) tables are not being generated

This suggests that the framework is not identifying any domain classes. Two possible reasons for this are:

- Domain class namespaces beginning with `NakedObjects`. The Naked Objects framework assumes that any classes within this namespace are part of the framework, and ignored by the domain model reflector. This is to avoid accidentally treating framework classes as domain entities, and thus building them into the database schema. Choose another namespace for your domain models.

- Domain classes not reachable. Naked Objects builds its metamodel (which in turn is passed to Entity Framework to create the entity model and hence the database schema) by walking the graph from known start points - which means the set of registered services. So if there is no way to get to a particular class, either directly or indirectly, via any method on any registered service - then the class won't have been reflected on during the start-up phase.

Unexpected behaviour in the user interface

This section contains suggestions on what to look for if your application is not behaving as you expect at the user interface.

A public method is not appearing as an object action

Possible causes:

- Naked Objects does not recognise any overloaded methods as actions. (This is because the various overloaded versions would show up on the menu with the same label.)
- Naked Objects does not recognise templated methods as actions.
- Naked Objects only recognises as actions those methods with parameter types that are either recognised value types or domain objects.

Default, Choices, Validate or other complementary methods are showing up as menu actions

If you have written a `Default`, `Choices`, `Validate` or other 'complementary method' that is intended to work with a Property or Action, and it shows up as a menu action on the user interface then the method isn't being recognised as a complementary method by the Naked Objects framework. This is probably due to one or more of the following:

- The name of the helper method (following the prefix) does not exactly match the name of the Property or Action it is intended to assist. The difference might be in spelling or case.

- The parameters of the helper method differ in type from those of the Property or Action it is intended to assist.

- A complementary method in a sub-class will only be applied to a property or action defined in a super-class, if that complementary method is also defined on the super-class and overridden in the sub-class. To put that another way: complementary methods must be defined in the same class as the property or action they apply to, but may then be overridden in sub-classes.

- The name of the corresponding action happens to begin with one of the prefixes recognised by Naked Objects (for example: `ClearComment`). The best policy is to avoid using action names that begin with this prefixes - give the action a different name and then use the `DisplayName` attribute to change it back to what you want on the user interface.

Debugging

This section deals with debugging your code.

Life Cycle methods are not being called

If you suspect that a life-cycle method is not being called, then you should first verify this by inserting a break point on the method. If it is not being called as expected, then the most likely explanations are:

- The method is not `public`.

- The method is not `void`.

- The method name is mis-spelled

- The method name does not begin with an initial upper-case letter.

The injected Container is null

The most likely explanation for this is that you have accidentally shadowed the property for the injected container in your domain class hierarchy. You can set Visual Studio to warn against this.

An injected service is null

This may be for the same reason as the point above. If not, then the most likely explanation is that you haven't registered the service.