**Naked Functions**

**Application Developer Manual**

**[under construction]**

Contents

[Introduction 4](#_Toc81411640)

[What is Naked Functions? 4](#_Toc81411641)

[Starting from the Naked Functions Template solution 5](#_Toc81411642)

[Client solution 10](#_Toc81411643)

[Writing your own applications 10](#_Toc81411644)

[Defining domain types 12](#_Toc81411645)

[Persisted domain types 12](#_Toc81411646)

[Properties 13](#_Toc81411647)

[Defining domain functions 14](#_Toc81411648)

[System services & configuration 15](#_Toc81411649)

[Authentication 15](#_Toc81411650)

[Authorization 15](#_Toc81411651)

[Auditing 15](#_Toc81411652)

[Profiling 15](#_Toc81411653)

[I18N 15](#_Toc81411654)

[Attributes 16](#_Toc81411655)

[Bounded 16](#_Toc81411656)

[CreateNew 16](#_Toc81411657)

[DefaultValue 17](#_Toc81411658)

[DescribedAs 17](#_Toc81411659)

[Disabled 18](#_Toc81411660)

[DisplayAsProperty 18](#_Toc81411661)

[Edit 19](#_Toc81411662)

[Hidden 19](#_Toc81411663)

[Mask 19](#_Toc81411664)

[MaxLength 20](#_Toc81411665)

[MemberOrder 20](#_Toc81411666)

[MinLength 20](#_Toc81411667)

[MultiLine 21](#_Toc81411668)

[Named 21](#_Toc81411669)

[Optionally 22](#_Toc81411670)

[PageSize 22](#_Toc81411671)

[Plural 23](#_Toc81411672)

[PresentationHint 23](#_Toc81411673)

[RegEx 24](#_Toc81411674)

[RenderEagerly 24](#_Toc81411675)

[TableView 25](#_Toc81411676)

[ValueRange 25](#_Toc81411677)

[ViewModel 26](#_Toc81411678)

[Versioned 26](#_Toc81411679)

# Introduction

## What is Naked Functions?

Naked Functions is a framework for developing enterprise-scale business applications for the .NET platform, using Microsoft’s Entity Framework Core to manage persistence on a relational database.

Naked Functions is different from most other .NET application development frameworks in two major respects:

1. All your application domain code, typically written in C#, follows a *pure* ‘functional programming’ patterns.
2. You do not need to write any user interface code at all: Naked Functions comes with a generic user interface that makes all the data and functionality of your application available to the user automatically. It is possible to customise this generic UI, which is written in TypeScript using the Angular framework and following standard Angular patterns, to any extent that you wish – however, you might be surprised by the effectiveness of the generic UI without any customisation at all.

Expanding on the first of these points, Functional programming (FP) means building programs from pure, side-effect free functions. While there are programming languages purpose-designed to support FP – such as Haskell, or F# – it is also possible to adopt a pure FP approach in ‘mixed-paradigm’ programming langages such as C#. The benefits of FP include:

* Testability. Because the result returned by a pure function depends only upon the arguments passed into it, and because the function generates no side-effects, automated tests are easier to write and more effective.
* Provability. If functions A and B are pure, side-effect free functions, and both A and B are correct, then any combination of A and B is also correct. This is not true when combining functions and methods that do not adopt this pure approach.
* Parallelism. Functionality written using the pure FP approach is much easier to parallelise for performance and scaleability.

However, there is a fundamental conundrum in FP, elegantly articulated by Simon Peyton Jones, one of the leading lights in the world of FP:

*‘The whole point of running a program is to have some side effect’ ­*  – whether this means writing to the database, sending an email, or even just displaying data on a screen.

The solution to this conundrum – expressed in non-technical terms – is that any real system will include both pure functions, which transform data, and ‘dirty’ functions that handle the input-output. Dirty functions may invoke pure functions, but not vice versa: if a pure function were to call a dirty function, then it becomes a dirty function itself. The traditional objective is to keep the dirty functions to an absolute minimum, but this is easier said than done. This is one of the reasons why, while FP is now widely adopted in the world of mathematical and scientific programming, it has made very little impact on traditional enterprise business applications, where there is typically far more code devoted to input/output than to pure computation.

Naked Functions addresses this challenge in a unique way: you *only* write pure functions, because all of the input/output (i.e. all of the interaction with the user, and the database) is handled by the framework, invisible to the application programmer. And, critically, your application functionality *never* makes calls into Naked Functions framework; it is the framework that makes calls into your application functions. The best way to understand how this is even possible is to look at a simple example.

## Starting from the Naked Functions Template solution

The Naked Functions Template provides a very simple application example than might be considered as a start point for building a full-scale enterprise application: it contains only a single record type – Student.

But don’t be fooled into thinking that Naked Functions is intended for the kind of simple applications you could have built using Microsoft Access! Naked Functions is designed specifically for building large-scale, highly-complex enterprise applications that may involve literally thousands of record types and millions of instances (corresponding to rows in the database), and with tens of thousands of functions defining business logic. It has been developed by the same team that designed the proven Naked Objects framework which has been in continuouos development for more than 20 years, and which has been deployed at very large scale. Naked Functions has some similarities to Naked Objects. The two frameworks share a common core and a common client, but adopt radically different application programming approaches: the former adopting pure object-oriented programming, and the latter adopting pure functional programming.

You can find the template solutions here:

<https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Template>

You will need to download two separate solutions from within this directory:

* The Naked Functions solution (note that there is also a Naked Objects Server within the same main directory – but we won’t be using this here).
* The SPA Client (SPA = ‘Single Page Application’)

To use these template solutions you will need Visual Studio 2019 or later, set up to work (initially) with the SQL Server LocalDb, which is a common start-point for prototyping applications using Entity Framework Core (you can switch to full SQL Server, or another compatible database at a later point).

Open the Template.Server.Sln in Visual Studio and run it. If your Visual Studio is set up correctly, then after a short delay – during which Entity Framework Core is creating and populating the initial version of the database – you should see a browser open on http://localhost:5000/ (we recommend using Chrome as the default) showing a page of JSON something like this:



If your browser does not have a JSON viewer installed, the result might be unformatted and harder to read, but this is not important: *you do not need to read or navigate this view, and it it is never seen by a real user in a deployed application .* This is just a direct view of the RESTful API that is generated by the Naked Functions framework based on the application model.

Now, *in a separate instance of Visual Studio*, open and run the Template.Client.Sln, which should launch another browser instance, this time on http://localhost:5001/, showing the Home screen of the generic SPA client. Once the client and server have connected – there might be a short delay here – the Home screen will display a ‘main menu’ called Students, providing actions for retrieving existing records and creating new ones.

Click on the Students menu (a real application will usually have multiple such menus side by side), then the All Students action, and from the returned list (of just three students) click on the first one, to view the very simple record for student Alie Algol:

Graphical user interface

Description automatically generated with medium confidence

There’s not much to explore here, yet, but try the following. Use the ‘back’ icon at the bottom of the screen (you may use the browser’s back button, here, but it is safer to user the icon) to return to the list of students. This time *right-click* on a Student and see what difference that makes. Then explore the other generic icons at the bottom of the screen.

Note that the student record has an Actions button, but this is greyed out because we don’t yet have any actions (functions) that may applied to a specific student, for example to update the record or generate an instance of another type of record related to this one (a TestScore, for example). The Reload button is like a safe version of ‘refresh’ but it is useful only when you might have multipler users updating the same data.

Let’s now explore the structure of this tiny application.

The server solution comprises two projects.

The Template.Model project holds all the application code. Importantly, it does *not* depend upon the Naked Functions framework. It does have a reference to the NakedFunctions.ProgrammingModel NuGet package, but this largely consists of attributes, plus a few interfaces. (In a more substantial application, the application model might be spread across multiple model projects.)

This model defines the Student type. For further information see Defining domain types.

The actions rendered on the Students menu are defned as pure functions, here:

[Named("Students")]

public static class Student\_MenuFunctions

{

public static (Student, IContext) CreateNewStudent(string fullName, IContext context)

{

var s = new Student { FullName = fullName };

return (s, context.WithNew(s));

}

public static IQueryable<Student> AllStudents(IContext context) =>

context.Instances<Student>();

public static IQueryable<Student> FindStudentByName(string name, IContext context) =>

context.Instances<Student>().Where(c => c.FullName.ToUpper().Contains(name.ToUpper()));

public static Student FindStudentById(int id, IContext context) =>

context.Instances<Student>().FirstOrDefault(c => c.Id == id);

}

Notes:

* All functions recognised by Naked Functions are static, and should be defined on static classes.
* By default actions seen by the user are formatted versions of the names of the functions, and the menu name is a formatted version of the name of the static class. Either may optionally be overridden using the Named attribute.
* The retrieval functions return either a single instance (e.g. FindStudentById) or some collection of a specified record type. If the type is specified as an IQueryable<T> (e.g. AllStudents), and the database has many matching records, then the results will be presented to the user in pages, automatically.
* All these menu functions take a parameter of type IContext. When the user invokes the action, Naked Functions will automatically pass in an implementation of IContext. This provides access to instances of any specified record type, to which LINQ may be applied. As you might guess, the context is acting as a wrapper onto the DbContext class, but it has other capabilities.
* If a function takes other parameters besides the IContext, then invoking it via the UI will display a dialog, providing fields for each parameter (except the IContext).
* The CreateNewStudent function returns a tuple. The first item in the tuple (a Student) will be displayed to the user. The second item (an IContext) alerts NakedFunctions that there are new and/or updated records that must be written to the database. The IContext itself is deliberately *immutable*, but the WithNew method will return a new copy of the original context, containing the newly created instance. Importantly, *the* CreateNewStudent *function does not update the database*. Rather, once the CreateNewStudent function has completed, NakedFunctions will handle the persistence of any new or updated objects to the database. The CreateNewStudent function is *pure* (it depends only on arguments passed in as parameters) and *side-effect free* – it returns new instances but does not mutate any existing instances or other aspects of the system.

The ExampleDbContext follows the standard pattern for any application written to work with Microsoft Entity Framework Core. It is best to define a DbSet for each persisted type e.g.:

public DbSet<Student> Students { get; set; }

Another option, useful during early stage prototyping is to define ‘seed data’, thus:

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<Student>().HasData(new Student { Id = 1, FullName = "Alie Algol" });

modelBuilder.Entity<Student>().HasData(new Student { Id = 2, FullName = "Forrest Fortran" });

modelBuilder.Entity<Student>().HasData(new Student { Id = 3, FullName = "James Java" });

}

The Template.Server project, which is the start-up project, runs the Naked Functions framework and generates the RESTful API based on the ‘model’ project(s) that it references. In the early stages of prototyping, there is no need to make any changes to the server project when you add or update types or functions to the model. (When your application becomes ready for deployment, then you will need to configure various capabilities in the server project).

The Startup class in the server project, which follows standard Microsoft patterns, sets up the NakedFunctions. It also needs to be told various things about the model:

* A list of application domain types (records)
* A list of all functions that need to be accessed via the UI (main menu functions and functions that will be rendered as actions on a given domain type).
* A separate list of the static types that define the main menus.
* A function that, whenever called, will create an instance of the DbContext for use by Entity Framework Core.

All of this information may be specified manually, but the Template solution uses a more convenient approach, whereby all four responsibilities are delegated to functions defined on ModelConfig::

public void ConfigureServices(IServiceCollection services)

{

...

services.AddNakedFramework(frameworkOptions =>

{

frameworkOptions.MainMenus = MenuHelper.GenerateMenus(ModelConfig.MainMenus());

frameworkOptions.AddEFCorePersistor(peristorOptions => { peristorOptions.ContextCreators = new[] { ModelConfig.EFCoreDbContextCreator }; });

frameworkOptions.AddNakedFunctions(appOptions =>

{

appOptions.FunctionalTypes = ModelConfig.Types();

appOptions.Functions = ModelConfig.Functions();

});

frameworkOptions.AddRestfulObjects(\_ => { });

});

ModelConfig is defined and maintaned in the model project.

public static class ModelConfig

{

public static Type[] Types() =>

DomainClasses.Where(t => t.Namespace == "Template.Model.Types" && t.IsStaticClass()).ToArray();

public static Type[] Functions() =>

DomainClasses.Where(t => t.Namespace == "Template.Model.Functions" && t.IsStaticClass()).ToArray();

public static Type[] MainMenus() =>

Functions().Where(t => t.FullName.Contains("MenuFunctions")).ToArray();

public static Func<IConfiguration, DbContext> EFCoreDbContextCreator =>

c => {

var db = new ExampleDbContext(c.GetConnectionString("ExampleCS"));

db.Create();

return db;

};  
 ...

}

Note that the first three functions make use of reflection (via helper methods DomainClasses and Functions) to create the lists automatically based on various optional naming conventions.

#### Server solution

Sets up the system using standard Microsoft patterns

Specifies where to find what is needed from the domain model programs.

Link to system service configuration section

### Client solution

The only things you *need* to do.

Things you *may* do, but with the *caveats* that you shouldn’t unless you need to. And you can keep the two teams very separate.

You would then, however, need to have skills in using Angular. For this reason separate manual for configuring and customising the Naked Objects Client.

## Writing your own applications

Define domain types

Define domain functions

Add persistent domain types to DbContext, and specify mapping if required

Register domain types with Naked Functions, manually or reflectively

Register all domain functions with NakedFunctions, manually or reflectively

Register types that define main menu functions, Register all domain functions with NakedFunctions, manually or reflectively

# Domain types

An application written to work with Naked Functions will typically define many domain types, in two varieties: persisted domain types and view models (which are never persisted). Note that both varieties may be displayed on the user interface. While it is possible to have one or more view models associated with each persisted type, the recommended approach is to display persisted types directly whenever possible (this keeps the code much simpler), bearing in mind that:

* Users can right-click on any associated object in order to display its details alongside the primary object
* Authorization may be used to control which properties may be seen by which users.

ViewModels are therefore recommended only when it is necessary to construct a view built from more than persisted object.

## Persisted domain types

Persisted domain types are mapped to the database by Entity Framework Core, either relying on conventions, or by means of explicit mapping. *Typically,* each such domain type corresponds to one database table.

Persisted domain types are *immutable*. It is possible to use immutable *classes*, but – for C# 9 at least – the recommended approach is to use *records*, because these support the C# with keyword that allows making a copy of an existing instance with changes to specified properties only. (Future versions of C# may eliminate the remaining distinctions between records and immutable classes).

Persisted domain types may be marked up with any of these attributes: Bounded, DescribedAs, Named, Plural, PresentationHint, RenderEagerly (for details of the effect of each, follow the link).

Each type must override the default implantation of these two methods, using this boilerplate code:

public override int GetHashCode() => base.GetHashCode();

public virtual bool Equals(Student other) => ReferenceEquals(this, other);

Note that his requirmement is not imposed by Naked Functions but by Entity Framework Core, to support lazy loading, which is in turn required to allow the user to link directly to any associated domain instances on the UI. (This is also the reason why properties must be virtual).

Domain types may optionally override the default ToString method in order to define the title for an instance on the UI – typically based on one or more of the properties, perhaps with additional text or formatting, for example:

public override string ToString() => $"{Department} {StartDate.ToString("d")}";

### Properties

Each domain type will usually have multiple properties, each property *typically* mapping to a column of the corresponding database table.

* One property (sometimes more than one) must define a unique Id (key field), following the normal rules for Entity Framework.
* All properties should be virtual and define {get; init;} accessors.
* Properties may be of any recognised value type, or of any persisted domain type, or an ICollection<T> where T is a persisted domain type. Collection properties must be initialised in code, for example:

public virtual ICollection<EmailAddress> EmailAddresses { get; init; } = new List<EmailAddress>();

* Properties may be marked up with any of these attributes: DescribedAs, Hidden, Mask, MemberOrder, MultiLine, Named, PresentationHint, Versioned (for details of the effect of each, follow the link).
* Collection properties may additionally be marked up with these attributes: RenderEagerly, TableView,

# Domain functions

## Introduction

The domain functionality for a Naked Functions application is implemented as functions. Most of these functions relate directly into user actions. For a function to constitute a user-action, or to provide supporting behaviour for a user action (the latter are the so-called complementary functions, which we will learn about later) the functons *must*:

* be public, static, and defined on static class.
* depend only on values passed in as parameters to the function.
* be side-effect free – i.e. not modify any of the passed-in parameter values, nor make any change to the system. They may, however, create new instances and return these as part of the result.
* define parameters using only recognised value types, domain types, and/or a single IContext.
* return a domain type, a collection of domain types, an IContext, or a tuple made up from these.
* be registered with the Naked Functions framework (see Registering functions).

Many such functions can be written in C# using expression syntax. For example:

public static IQueryable<Product> FindProductByName(string match, IContext context) => context.Instances<Product>().Where(x => x.Name.ToUpper().Contains(match.ToUpper()));

public static string ValidateUpdateDateOfBirth(  
 this Employee e, DateTime? dateOfBirth, IContext context) =>

(dateOfBirth > context.Today().AddYears(-16)) ||   
 (dateOfBirth < context.Today().AddYears(-100)) ? "Invalid Date Of Birth" : null;

However, the use of expression syntax is not a requirement for Naked Functions. While C# now contains many features appropriate to FP, it currently lags pure FP languages, so you may sometimes find it necessary – or just simpler – to write function bodies using multiple statements. For example:

public static StaffSummary GenerateStaffSummary(IContext context)

{

var staff = context.Instances<Employee>();

int female = staff.Where(x => x.Gender == "F").Count();

int male = staff.Where(x => x.Gender == "M").Count();

return new() { Female = female, Male = male };

}

### Registering functions

Every function that is intended to be a user-action, or to provide supporting behaviour for a user action the functons must be registered with the Naked Functions framework. (It is not necessary to register functions that are only ever called *indirectly* from user actions or their complementary functions). A useful convention is to make such functions internal rather than public.)

Registration is done within the ConfigureServices method of the standard .NET Startup class in the server project (where domain types are also registered):

public void ConfigureServices(IServiceCollection services)

{

...

services.AddNakedFramework(frameworkOptions =>

{

...

frameworkOptions.AddNakedFunctions(appOptions =>

{

appOptions.DomainTypes = new Type[] { };

appOptions.DomainFunctions = new Type[] { add static types defining functions here};

});

The appOptions.DomainFunctions Type[] – which should include all the static types on which domain functions have been defined (any non-public functions defined on those types will be ignored).

This list may be defined manually in code, or you may choose to use a combination of namespace/naming conventions and reflection to construct the list automatically. For example, in the Template.Server project the creation of the list is delegated to a ModelConfig class:

appOptions.DomainFunctions = ModelConfig.TypesDefiningDomainFunctions();

which then finds all static classes in the Template.Model.Functions namespace:

public static class ModelConfig

{

public static Type[] TypesDefiningDomainFunctions() =>

PublicClassesInterfacesEnums.Where(t => t.Namespace == "Template.Model.Functions"   
 && t.IsStaticClass()).ToArray();

private static IEnumerable<Type> PublicClassesInterfacesEnums =>

typeof(ModelConfig).Assembly.GetTypes().Where(t => t.IsPublic && (t.IsClass ||   
 t.IsInterface || t.IsEnum));

private static bool IsStaticClass(this Type t) => t.IsAbstract && t.IsSealed;

### Types of function

Registered domain functions are of four fundamental types:

* Main menu functions
* Instance functions
* Collection-contributed functions
* Complementary functions

#### Main menu functions

Main menu functions appear as actions on the ‘main menus’ – shown on the Home page of the generic client. For example:

Text

Description automatically generated

Each main menu (Employees, Addresses …) is defined by a single static type. For example:

[Named("Customers")]

public static class Customer\_MenuFunctions

{

[MemberOrder(10)]

public static Customer FindCustomerByAccountNumber(...

[MemberOrder("Stores", 1)]

public static IQueryable<Customer> FindStoreByName(...

(See also Named and MemberOrder attributes)

*In addition to being registered with all domain functions* (see Registering functions) the static classes defining the main menus, must be registered with Naked Functions (in Startup# ConfigureServices) as main menus:

services.AddNakedFramework(frameworkOptions =>

{

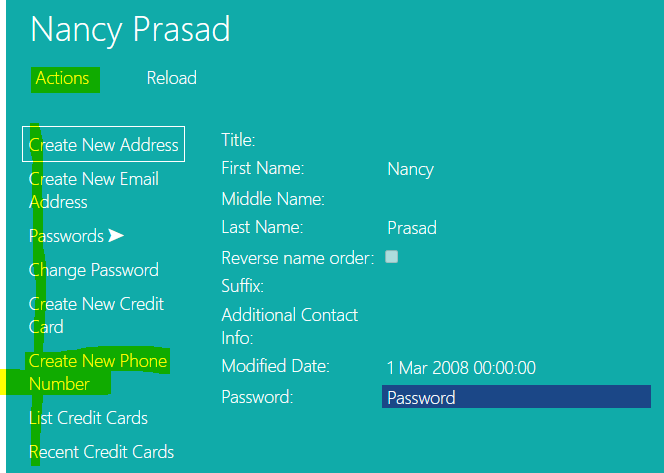
frameworkOptions.MainMenus = MenuHelper.GenerateMenus( array of types required );  
 ...

The required array of the main menu static classes may be created manually, or by using a combination of naming conventions and reflection, as in the Template.Server project, where it is delegated to a method on the ModelConfig class:

public static Type[] MainMenus() => TypesDefiningDomainFunctions ()  
 .Where(t => t.FullName.Contains("MenuFunctions")).ToArray();

#### Instance functions

Instance functions are associated with instances of a specified domain type. They appear on the generic client as actions on the Actions menu on the instance. For example:



Instance functions are define using the C# syntax for extension methods. For example:

public static IContext CreateNewPhoneNumber(this Person p,

PhoneNumberType type,

string phoneNumber,

IContext context)

When the user clicks on such an action, the first parameter (corresponding to the instance on which the action is appearing) is not rendered in the dialog (nor is the IContext), but the instance will be provided by the framework when the user invokes the action by clicking OK.

Graphical user interface, application

Description automatically generated

Note: Instance functions that define no other parameters (except, optionally, an IContext) are executed immediately when the user clicks on the action.

#### Query-contributed functions

Query-contributed functions are somewhat like instance functions, but they are applied to an IQueryable<domainType> rather than to a single instance of a domain type. For example:

public static IContext TerminateOffers(this IQueryable<SpecialOffer> offers,   
IContext context) =>

On the user interface, when the user is presented with a list of the corresponding type that has been returned (as an IQueryable) by another action, any matching query-contributed function will be made available via the Actions menu:

Text

Description automatically generated

The list of instances will be rendered with check boxes alongside. The user may select any sub-set of the list and invoke the action (which may also require other parameters to be specied, but in this example does not, so it will be executed immediately). The IQueryable passed into the function will now contain only those instances selected by the user.

#### Complementary functions

See Complementary functions.

## Retrieving instances

The IContext provides an Instances<T> method for accessing the instances of any persisted domain type as an IQueryable<T>, to which LINQ expressions may then be applied. By this means functions can remain ‘pure’ – accessing instances without calling upon the DbContext or any other systems capability.

This could have been implemented by passing in the DbContext automatically. However, that would make all functions dependent on a specific implementation of DbContext, and it would still require another type to provide the further capabilities that we shall introduce shortly.

Here is an example of how a function uses the IContext, along with other parameters, to find and return matching instances:

public static IQueryable<PurchaseOrderHeader> OpenPurchaseOrdersForProduct(

Product product, IContext context) =>

from obj in context.Instances<PurchaseOrderDetail>()

where obj.Product.ProductID == product.ProductID &&

obj.PurchaseOrderHeader.Status <= 2

select obj.PurchaseOrderHeader;

## Creating new instances

The IContext is also the mechanism by which a function can create or update persisted domain instances. Importantly, the function itself does not persist the new or updated object – that would break the ‘side-effect free rule’. Rather, it returns an IContext that contains a list of new instances that need to be persisted, or existing instances that need to be updated on the database. *After the domain function has exited*, Naked Functions will pick up this information from the returned IContext and make the necessary changes to the database, encapsulated within a transaction. The domain function is side-effect free.

But if the function had modified any of the domain instances passed in, or modified the IContext itself (to advise of the changes) this would also be a violation. Fortunately, neither of these is required, because:

* Instead of modifying an existing domain instance, the domain function will create a new instance that is a complete copy of the existing one – with, importantly, the same Id – but with the required differences. This is standard practice in functional programming, or working with immutable types generally. If your domain type is a C# record then this can be done, trivially, using the C# with operator.
* Instead of modifying the IContext that has been passed in as a parameter, the domain function invokes either the .WithNew() or the .WithUpdated() method on that context. These methods create a new instance of IContext, with all the previous details preserves but with new instances added to the list of either instances to be persisted, or instances to be updated. (It is quite possible to do this repeatedly, forming a chain of IContexts before the final one is returned from the function).

Here is an example of a domain function that creates a new instance based on details provided by the user:

[CreateNew]

public static (WorkOrder, IContext context) CreateNewWorkOrder(

Product product, int orderQty, DateTime startDate, IContext context)

{

var wo = new WorkOrder()

{

ProductID = product.ProductID,

OrderQty = orderQty,

ScrappedQty = 0,

StartDate = startDate,

DueDate = startDate.AddDays(7),

ModifiedDate = context.Now()

};

return (wo, context.WithNew(wo));

}

Notes:

* The function returns a tuple. The first item in the tuple – which may be a domain type or an ICollection<DomainType> - is displayed to the user. The second item is an IContext, which will contain details of any instance(s) that the framework must persist. In this example, the function both displays and persists the same instance, but that is not always the case.
* The .WithNew(wo) results in a new IContext capturing the instance (which will not have its Id properties set, if the Ids are database managed)
* The use the CreateNew attribute is optional, but it has a benefit at the user interface. See CreateNew.
* Context.Now() is explained here.
* The WorkOrder displayed to the user when the framework has completed its work will in fact be retrieved back from the database after it has been persisted – so it will now have a proper Id.

## Updating existing instances

Here is an example of a domain function that updates an existing instance:

public static IContext AppendComment(

this SalesOrderHeader order, string commentToAppend, IContext context)

{

string newComments = order.Comment == null ? commentToAppend : order.Comment + "; " + commentToAppend;

SalesOrderHeader updated = order with { Comment = newComments, ModifiedDate = context.Now() };

return context.WithUpdated(order, updated);

}

Notes:

* .WithUpdated requires both the original instance and the updated instance (created from the original using with) – so that Naked Functions can associate them.
* This function returns only an IContext, not a tuple. The function is an extension method, meaning that it will have been invoked via the actions menu on the instance of SalesOrderHeader that constitutes the this parameter. Returning only an IContext indicates that after the framework has completed the updates, the user interface should display the updated version of the instance that the user was viewing previously. The tuple would be needed only if it was desired to display a new or different instance.

Here is an example of a more complex function that updates multiple instances passed in as an IQueryable (see Collection-contributed functions)

public static IContext ClearCommentsFromOrders(this IQueryable<SalesOrderHeader> toOrders, IContext context)

{

var updates = toOrders.Select(x => new { original = x, updated = WithClearedComments(x, context) });

return updates.Aggregate(context, (c, of) => c.WithUpdated(of.original, of.updated));

}

## IContext

In functional programming, functions must be:

* ‘pure’ – they may not access any systems functionality except as passed in as a parameter.
* ‘side-effect free’ – they may not make changes external to the function, or make changes to anything passed in as a parameter. The result they return must either be an unmodified item access via the parameter values, or new instances.

The IContext is an important mechanism that allows *all* the domain functions that make up a Naked Functions application to abide rigidly to these two rules.

If a domain function that is registered with the framework (so it can be invoked as an action by the user) defines a parameter of type IContext, that parameter will not be visible on the user interface. But when the action is invoked by the user, Naked Functions will call the corresponding function and pass in an implementation of IContext automatically.

### Accessing the current date/time

### Accessing the current user

### Creating a GUID

### Using random numbers

### Accessing system services

## Complementary functions

A ‘complementary function’ is a function that complements a domain function that is made available to the user as an action: it enriches the presentation and/or behaviour of that action, or of its parameters. Six forms of complementary function are recognised:

* Validate function
* Default function
* Choices function
* AutoComplete function
* Hide function (note that this one may complement an action *or* a property on a type)
* Disable

### Validate function

Simple forms of parameter validation may be implemented just by the addition of the MaxLength, MinLength, Optionally, RegEx, and/or ValueRange attributes. Where custom domain rules must be applied, use a Validate complementary function.

A Validate method returns a string. If the parameter value(s) pass the validation test a null or empty string should be returned. If the validation tests fail, a non-empty string should be returned, containing a validation-fail message that will be presented to the user. For example:

public static IContext EditQuantities(this SpecialOffer sp,

int minQty, int maxQty, IContext context) => ...

public static string ValidateEditQuantities(

this SpecialOffer sp, int minQty, int maxQty, IContext context) =>

maxQty != null && maxQty.Value < minQty ? "Max Qty cannot be < Min Qty" : null;

Notes:

* The Validate function must be defined on the same static class as the function that it complements. It has the same name (with the same casing) as the function it complements, but prefixed by ‘Validate’. It also takes the same parameter list. (Any parameter attributes do not need to be reproduced in the Validate function, and will be ignored if they are).
* The IContext may be used within the function, for example to retrieve other instances.
* The Validate function will be called when the user hits the OK button on the dialog, and with all the values automatically supplied by the system. If validation fails the main function will not be invoked, and the validation-fail message will be rendered to the user. For example: The Validate function must be defined on the same static class as the function that it complements. It has the same name (with the same casing) as the function it complements, but prefixed by ‘Validate’. It also takes the same parameter list. (Any parameter attributes do not need to be reproduced in the Validate function, and will be ignored if they are).
* The IContext may be used within the function, for example to retrieve other instances.

Graphical user interface, text, application

Description automatically generated

A Validate function may be applied to a single parameter of a function. As well as breaking up the logic, where appropriate, this also allows the validation message to be presented adjacent to the incorrect paremeter, rather than at the end of the list. For this case, the Validate *prefix* is immediately followed by the parameter number (counting from zero), and the parameter list has only the first parameter of the corresponding function *if that function is contributed to a type,* plus the parameter of interest (and, *optionally*, an IContext). And individual parameter Validate function may be used in conjunction with the all-parameter Validate function. For example, the following function can be added into the example above to check that the minimum quantity is always greater than 0:

public static string Validate1EditQuantities(this SpecialOffer sp, int minQty) =>

minQty < 1 ? "Must be > 0" : null;

Graphical user interface, application

Description automatically generated

### Default function

A hard-coded default value for a value-type parameter may be specified using the DefaultValue attribute. To specify a default value that varies, or to specify a default value for a domain-type parameter, use a Default complementary function. For example:

public static IQueryable<Product> ListBikes(ProductCategory category, ProductSubcategory subCategory, IContext context) =>

public static ProductCategory Default0ListBikes(IContext context) =>

context.Instances<ProductCategory>().Skip(1);

Notes:

* The Default function must be defined on the same static class as the function that it complements. It has the same name as the function that it complements, prefixed by ‘Default’ followed by the parameter number (counting from zero) to which the default value applies.
* The return type should match the type of the specified parameter number on function it complements.
* If the complemented function is an ‘extension method’- i.e. contributed to the actions menu on a type – (the example above is not; it is a main menu function) then the Default function should also be defined as an extension method to the same type. An IContext parameter may optionally be added, if required.

### Choices function

Where a domain type is known to have a limited number of instances, the Bounded attribute may be used, in which case any parameter of that type will automatically present the instances as a drop-down list. An enum type parameter will also automatically be rendered as a drop-down list. To specify a custom set of choices from the instances of a persisted domain type, or choices for a value type parameter, use a Choices complementary function. In the example below, the EditStateProvince function has two complementary Choices functions:

public static IContext EditStateProvince(this Address a,

CountryRegion countryRegion, StateProvince stateProvince, IContext context) => ...

public static IList<CountryRegion> Choices1EditStateProvince(this Address a, IContext context) => context.Instances<CountryRegion>().ToArray();

public static IList<StateProvince> Choices2EditStateProvince(this Address a,

CountryRegion countryRegion, IContext context) =>

context.Instances<StateProvince>().Where(p => p.CountryRegion.CountryRegionCode == country.CountryRegionCode).OrderBy(p => p.Name).ToArray()

* The Choices function must be defined on the same static class as the function that it complements. It has the same name as the function that it complements, prefixed by ‘Choices followed by the parameter number (counting from zero) to which the value choices apply.
* The return type should be an IList<T> where T is type of the specified parameter number on function it complements.
* If the complemented function is an ‘extension method’- i.e. contributed to the actions menu on a type – then the Choices function should also be defined as an extension method to the same type. An IContext parameter may optionally be added, if required.
* If the choices for the parameter depend on the value entered by the user for another parameter, then that parameter may also be added to the Choices function signature (both the name and type must match) and used in the logic. This is the case in the second example above, where the choices for the State Province depend upon the Country Region chosen.
* Choices may be used in conjunction with a Default function, in which case the programmer should ensure that the default value is always one of the specified set of choices.

### AutoComplete function

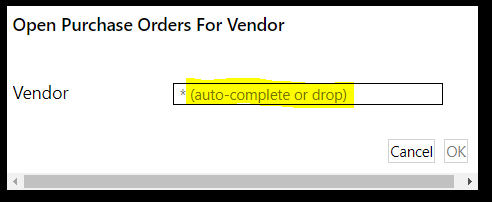
The user may specify the value for a domain type parameter by selecting from the choices (if specified – see Choices function), or by dragging and dropping a reference from elsewhere on the screen (including a second pane). A third option is to find the required instance by auto-complete, implemented through an AutoComplete complementary function. For example:

public static IQueryable<PurchaseOrderHeader> OpenPurchaseOrdersForVendor(Vendor vendor, IContext context) => ...

[PageSize(10)]

public static IQueryable<Vendor> AutoComplete0OpenPurchaseOrdersForVendor([MinLength(2)] string name, IContext context) =>

context.Instances<Vendor>().Where(v => v.Name.ToUpper().StartsWith(name.ToUpper()));



Graphical user interface, text, application

Description automatically generated

Notes:

* The AutoComplete function must be defined on the same static class as the function that it complements. It has the same name as the function that it complements, prefixed by ‘AutoComplete’ followed by the parameter number (counting from zero) to which the auto-complete is applied.
* The return type should be an IQueryable<T> where T is type of the specified parameter number on function it complements.
* If the complemented function is an ‘extension method’- i.e. contributed to the actions menu on a type – then the AutoComplete function should also be defined as an extension method to the same type.
* The AutoComplete function should offer a single string parameter, representing the characters typed in by the user so far. An IContext parameter may optionally be added – and this will usually be needed to access domain instances.
* The use of MinLength and PageSize attributes is optional.

### Hide function

Properties that are never intended to be visible to a user may be hidden with the Hidden attribute. Also properties, or actions may be hidden from certain users or roles using Authorization. Less commonly, it may be deemed appropriate to hide specific properties or actions on a variable basis – for example based on the state of an instance. For this a Hide complementary function may be used.

1. Hide may be applied to a function. For example:

public static IContext Approve(this PurchaseOrderHeader po, IContext context) => ...

public static bool HideApprove(this PurchaseOrderHeader po) => !po.IsPending();

Notes:

* The Hide function must be defined on the same static class as the function that it complements. It has the same name (with the same casing) as the function it complements but prefixed by ‘Hide.
* If the complemented function is an ‘extension method’- i.e. contributed to the actions menu on a type (as in the above example) – then the Hide function should also be defined as an extension method to the same type.
* An IContext may be added if needed (it is not needed in the example above, where the implementation uses date/functionality available on the PurchaseOrderHeader.

2. Hide may also be applied to a property. In the following example, two Hide methods apply to the Store and Person *properties* of a Customer type, depending on the type of the customer:

public static bool HideStore(this Customer c) => !IsStore(c);

public static bool HidePerson(this Customer c) => !IsIndividual(c);

* To hide a property the Hide complementary function must be defined as an extension method for the type holding the property.
* The name of the Hide function should be the name of the property (in code – not as presented to the user) prefixed by ‘Hide’.

### Disable function

Any action that is visible to the current user, may be temporarily disabled – for example based upon the state of an instance, or associated instances – through a Disable complementary function.

1. Disable is typically applied to a function. For example:

public static IContext RemoveDetail(this SalesOrderHeader soh,

SalesOrderDetail detailToRemove, IContext context) =>

public static string DisableRemoveDetail(this SalesOrderHeader soh) =>

soh.Details.Any() ? null : "Order has no Details.";

Notes:

* The Disable function must be defined on the same static class as the function that it complements. It has the same name (with the same casing) as the function it complements but prefixed by ‘Disable’.
* If the complemented function is an ‘extension method’- i.e. contributed to the actions menu on a type (as in the above example) – then the Disable function should also be defined as an extension method to the same type.
* Disable takes no other parameters, except for an optional IContext if needed.
* Disable returns a string. If the returned string is null or empty, the action remains enabled. If the returned string is non-empty, the action will be disabled and the generic client will render the returned string as a tooltip such that the user may identify the reason why the action is disabled.
* It is recommended that a function should be disabled only if there is a possibility that it could become enabled again e.g. by further user actions. If the action will not become available again, for example due to the lifecycle status of the domain instance, then consider hiding it instead (see Hide function).

2. The Disable complementary function may also be used to disable a specific parameter in a manner similar to the Disabled attribute but using custom logic. However, disabling of parameters in a dialog should be a rare occurrence, and makes sense only if a valid value has already been set up using a DefaultValue attribute, or Default function.

# Attributes

## Bounded

Applied to a persisted domain type. Specifies that the type has few instances and these should be offered as drop-down choices in any action parameter of that type. For example:

[Bounded]

public record Department : IHasModifiedDate

...

public static IContext ChangeDepartmentOrShift(

this Employee e, Department department, Shift shift, IContext context)

Graphical user interface, text, application

Description automatically generated

## CreateNew

Applied to a function. Specifies that the instance returned as the first item in the tuple will be a new instance. Instead of rendering this function as a typical dialog, it will be rendered in a different way, where the fields are shown in the context of a persisted instance, but with the other fields rendered empty. For example:

[CreateNew]

public static (PurchaseOrderHeader, IContext) CreateNewPurchaseOrder(

Vendor vendor,

ShipMethod shipMethod,

IContext context)

{

var po = new PurchaseOrderHeader()

{

RevisionNumber = 0,

Status = (byte)POStatus.Pending,

VendorID = vendor.BusinessEntityID,

ShipMethodID = shipMethod.ShipMethodID,

OrderDate = context.Today(),

ModifiedDate = context.Now()

};

return (po, context.WithNew(po));  
}

Graphical user interface, text, application

Description automatically generated

## DefaultValue

Applied to an integer parameter on a function definition. Specifies default value for parameter. If used on a DateTime, an integer value indicates a day relative to today e.g. -1 means 'yesterday'. For example:

public static SalesOrderHeader FindOrder([DefaultValue("SO")] string orderNumber, IContext context) =>

Text

Description automatically generated

## DescribedAs

Applied to Domain Types, Functions, Parameters. Specifies a (short) descriprition, or help, to be rendered as a 'tooltip' or 'placeholder' on the UI, according to the context. For example:

[DescribedAs("... from an existing Employee")]

public static SalesPerson CreateNewSalesPerson( Employee employee) {

Graphical user interface, application, Word

Description automatically generated

## Disabled

Applied to a parameter on a function definition. Renders parameter but does not permit the user to change that value (useful for providing advisory information). For example:

public static IQueryable<Product> ListBikes(

[Disabled] ProductCategory category, ProductSubcategory subCategory, IContext context)

Note that this is only of use if a value is being provided programmatically for the disabled parameter, for example via a Default function.

Graphical user interface, text

Description automatically generated

## DisplayAsProperty

Applied to a read-only type-contributed function that returns a value, reference, or collection. Specifies that the function should be called whenever the type is displayed, and the results of calling the function rendered as a property on the type. For example:

[DisplayAsProperty]

public static ICollection<SpecialOffer> SpecialOffers(this IProduct product, IContext context)

{

int pid = product.ProductID;

return context.Instances<SpecialOfferProduct>().Where(sop => sop.ProductID == pid).Select(sop => sop.SpecialOffer).ToList();

}

A picture containing timeline

Description automatically generated

## Edit

Applied to a function intended solely to update the values of one or more properties on a single instance. The generic UI will then render this function as an edit icon next to the property (or properties) and overlay the dialog on top of the existing property rather than separately – to give the appearance of editing the property itself. For example:

[Edit]

public static IContext UpdateNationalIDNumber(this Employee e,

[MaxLength(15)] string nationalIdNumber, IContext context) =>

UpdateEmployee(e, e with { NationalIDNumber = nationalIdNumber }, context);

Graphical user interface, application

Description automatically generated Graphical user interface, application

Description automatically generated

## Hidden

Applied to a property. Specifies that a property should never be shown on the UI (irrespective of the user’s authorization).

## Mask

Applied to a value-type property. Specifies that the value should be formatted using standard Microsoft formats. For example:

[Mask("d")]

public virtual DateTime? DateOfBirth { get; init; }

Graphical user interface, text, application

Description automatically generated

## MaxLength

Applied to a string parameter. Specifies the maximum number of characters that may be entered. For example:

public static IContext UpdateNationalIDNumber(this Employee e,

[MaxLength(15)] string nationalIdNumber, IContext context) =>

UpdateEmployee(e, e with { NationalIDNumber = nationalIdNumber }, context);

Text

Description automatically generated

## MemberOrder

1. Applied to a property. Within the type, properties will be rendered in ascending order of MemberOrder value, followed by any members that have no MemberOrder specified.

2. Applied to a function. Specifies the ordering of actions within a menu. If the optional string grouping parameter is specified, this will result in the creation of an expandable/collapsable sub-menu, and the MemberOrder value will be applied within that sub-menu. For example:

[MemberOrder("Stores", 1)]

public static IQueryable<Customer> FindStoreByName(string name, IContext context)

Text

Description automatically generated

## MinLength

Applied to a string parameter. Specifies minumum accepted length. If used on the string parameter of an AutoComplete complementary function, specifies the minimum number of characters that must be typed before the auto-complete function will be engaged. For example:

public static IQueryable<PurchaseOrderHeader> OpenPurchaseOrders(this Vendor vendor, IContext context) =>

PurchaseOrder\_MenuFunctions.OpenPurchaseOrdersForVendor(vendor, context);

public static IQueryable<Vendor> AutoComplete0OpenPurchaseOrders(this Vendor vendor, [MinLength(2)] string name, IContext context) =>

PurchaseOrder\_MenuFunctions.AutoComplete0OpenPurchaseOrdersForVendor(name, context);

## MultiLine

1. Applied to a string property or parameter. Specifies that a the property/parameter should be rendered as a multi-line text field, with specified number of lines, scrollable. The Width property is unused by the current client, but is exposed on the Resftul API for custom use. For example:

[MultiLine(10)]

public virtual string Description { get; init; }

Graphical user interface, text, application

Description automatically generated

2. Applied to a function. Specifies that the user may invoke the corresponding action repeatedly - building up a table of entries. For example:

[MultiLine()]

public static IContext AddNewDetails(this PurchaseOrderHeader header,

Product prod, short qty, decimal unitPrice, IContext context) =>

Graphical user interface, text

Description automatically generated

## Named

Applied to any domain type, property, function, or parameter. Specifies that the default name rendered on the user interface (a reformatted version of the name used in the code) should be overridden. It is recommended that for simplicity the names should be the same in the code and on the display whenever possible, but the ability to override this is useful where there is a need for e.g. punctuation in the display version that cannot be used within the code name. For example:

public static (Address, IContext) CreateNewAddress(

AddressType type,

string line1,

string line2,

string city,

string postCode,

[Named("State / Province")] StateProvince sp,

IContext context)

Graphical user interface, application

Description automatically generated

## Optionally

Applied to a parameter. Specifies that the parameter is optional within a dialog. The client will not then render a '\*' in the field, nor require an entry in the field before the user may click 'OK'. For example:

public static IQueryable<Employee> FindEmployeeByName(

[Optionally] string firstName, string lastName, IContext context)



## PageSize

1. Applied to a function that returns IQueryable<T>. Overrides the default page size (of 20 instances) for the rendered results. For example:

[PageSize(10)]

public static IQueryable<Customer> FindStoreByName(string name, IContext context)

2. Applied to an AutoComplete complementary function. Specifies how many matching entries (max) will be offered to the user. For example:

public static IContext AssociateWithSpecialOffer(

this Product product, SpecialOffer offer, IContext context) =>

SpecialOffer\_Functions.AssociateWithProduct(offer, product, context);

[PageSize(10)]

public static IQueryable<SpecialOffer> AutoComplete1AssociateWithSpecialOffer(

this Product product,

[MinLength(2)] string name, IContext context) =>

context.Instances<SpecialOffer>().Where(specialOffer => specialOffer.Description.ToUpper().StartsWith(name.ToUpper()));

## Plural

Applied to a type. Where a type name needs to be rendered in plural form (for example on a collection) the type name is automatically pluralised following simple rules. Where the auto-generated plural name does not read correctly, the plural name may be specified explicitly using the Plural attribute. For example:

[Plural("Bills Of Material")]

public record BillOfMaterial : IHasModifiedDate

## PresentationHint

Applied to any type, property, function, or parameter. Allows the programmer to specify hints that can be picked up and applied by a customized user interface. The string specified in the PresentationHint, is passed through the RESTful API (as an "x-ro-nof-presentationHint":"") and in the generic client will be added as a class to the corresponding Html element. For example:

[PresentationHint("Foo")]

public record Location

Graphical user interface, text, application

Description automatically generated

## RegEx

Applied to a string parameter. Specifies that any entered string must conform to the regex pattern given. For example:

string cardType,

[RegEx("^[0-9]{16}$")][DescribedAs("No spaces")] string cardNumber,

[RegEx("^[0-9]{2}/[0-9]{2}")][DescribedAs("mm/yy")] string expires,

IContext context

)

Graphical user interface, application

Description automatically generated

Note that by also using DescribedAs, the user was prompted with the correct format on the empty field:



## RenderEagerly

(See also TableView)

1. Applied to a collection property, specifies that the collection should automatically be opened *as a list* when the instance is displayed. If the property also has a TableView attribute then the collection will automatically be opened in table view. For example:

[RenderEagerly, TableView(...)]

public virtual ICollection<ProductInventory> ProductInventory { get; init; } ...

2. Applied to a type, specifies that *all* collection properties on that type should automatically be opened when the instance is displayed. If any property also has a TableView attribute then that collection will automatically be opened in table view.

3. Applied to a function that returns a collection, specifies that the returned collection should be rendered in table view rather than list view. If the function also has a TableView attribute then the returned result will automatically be opened in table view.

## TableView

(See also RenderEagerly)

1. Applied to a collection property, allows the table view of the collection to be customized, showing which columns should be visible and in which order.

[TableView(false, nameof(Types.ProductInventory.Quantity), nameof(Types.ProductInventory.Location),

nameof(Types.ProductInventory.Shelf), nameof(Types.ProductInventory.Bin))]

public virtual ICollection<ProductInventory> ProductInventory { get; init; } = new List<ProductInventory>();

Chart, table, treemap chart

Description automatically generated

2. Applied to a function that returns a collection, specifies the formatting for the returned result, *when displayed as a table view.*

Notes (for both cases):

* The first (Boolean) property of the attribute specifies whether the title of the instance should also be rendered. For example.
* The names of the columns should be specified as the actual names of the properties in the type of the collection – not the names as presented on the UI (if different). The safest, and most convenient, way to ensure this is to use the C# nameof keyword as shown above.

## ValueRange

1. Applied to an integer property, specifies a minimum and maximum acceptable value. For example:

public static SpecialOffer BestSpecialOffer(

this Product p, [ValueRange(1, 999)] int quantity, IContext context) =>

1. Applied to a DateTime property, specifies the earliest and latest acceptable dates *relative to today* where negative values are before today, 0 means today, and positive values are after today. For example:

public static IContext AddProductReview(this Product p,

[DefaultValue(0), ValueRange(-30, 0)] DateTime dateOfReview,

The above code requires the date to be within the last 30 days, including today.

## Versioned

Applied to a property that is changed whenever the database row is updated. Used to test that the user's view of an object is up to date before allowing a function that returns an IContext may be invoked. For example:

[Versioned]

public virtual DateTime ModifiedDate { get; init; }

Note that this is effectively a 'long term' form of concurrency checking i.e. to test whether the user’s view is stale before any update is commenced. *Entity Framework's 'short term' form of concurrency checking, which checks for update contention during a transaction, should also be used*. The two mechanisms may use the same field, if the type is acceptable for Entity Framework Core, or use different fields for their checks. Responsibility for updating the value may be delegated to the database or implemented in code. (See also LifeCycle functions).

## ViewModel

Applied to a domain type, specifies that this is a view model, not a persisted type. For example:

[ViewModel(typeof(CustomerDashboard\_Functions))]

public record CustomerDashboard {

[Hidden]

public virtual Customer Root { get; init; }  
   
 //Other properties are derived from the Root in this example

The attribute must be given a static class that defines the two functions that specify how the string key(s) for this view model should be derived from the information within it, and how to re-create the view model instance using these keys (thereby allowing the user to navigate back and forth). These signature of these functions must follow this form:

public static string[] DeriveKeys(this VMT vm)

public static VMT CreateFromKeys(string[] keys, IContext context)

Where VMT is the view model type. For example:

public static class CustomerDashboard\_Functions {

public static string[] DeriveKeys(this CustomerDashboard cd) =>

new[] {cd.Root.CustomerID.ToString() };

public static CustomerDashboard CreateFromKeys(string[] keys, IContext context)

{

int customerId = int.Parse(keys[0]);

return new CustomerDashboard {

Root = context.Instances<Customer>().Single(c => c.CustomerID == customerId)};

}

}

Note that any view model must still be registered as a domain type.

# System services & configuration

General introduction to configuring the

## Authentication

## Authorization

## Auditing

## Profiling

## I18N