**Naked Functions**

**Application Developer Manual**

**[under construction]**

Contents

[Introduction 6](#_Toc82006784)

[What is Naked Functions? 6](#_Toc82006785)

[Starting from the Naked Functions Template 7](#_Toc82006786)

[Domain types 11](#_Toc82006787)

[Registering domain types 11](#_Toc82006788)

[Persisted domain types 12](#_Toc82006789)

[Domain Interfaces 13](#_Toc82006790)

[View Models 14](#_Toc82006791)

[Domain-specific enums 15](#_Toc82006792)

[How-to’s 15](#_Toc82006793)

[Domain functions 16](#_Toc82006794)

[Registering domain functions 17](#_Toc82006795)

[Main menu functions 18](#_Toc82006796)

[Instance functions 19](#_Toc82006797)

[Query-contributed functions 20](#_Toc82006798)

[Complementary functions 20](#_Toc82006799)

[Validate function 21](#_Toc82006800)

[Default function 22](#_Toc82006801)

[Choices function 23](#_Toc82006802)

[AutoComplete function 23](#_Toc82006803)

[Hide function 25](#_Toc82006804)

[Disable function 25](#_Toc82006805)

[How to’s 26](#_Toc82006806)

[How to retrieve instances from the database 26](#_Toc82006807)

[How to save new instances to the database 26](#_Toc82006808)

[How to update existing instances in the database 27](#_Toc82006809)

[How to access the current user 27](#_Toc82006810)

[How to access the current date/time 27](#_Toc82006811)

[How to use random numbers 27](#_Toc82006812)

[How to display feedback messages to the user 27](#_Toc82006813)

[How to generate a GUID 27](#_Toc82006814)

[How to get hold of a registered service 27](#_Toc82006815)

[How to implement a user action that has custom side-effects 27](#_Toc82006816)

[How to write a function to edit a property in-line 28](#_Toc82006817)

[Using the IContext 29](#_Toc82006818)

[IContext.Instances 29](#_Toc82006819)

[IContext.WithNew 30](#_Toc82006820)

[IContext.WithUpdated 31](#_Toc82006821)

[IContext.GetService 32](#_Toc82006822)

[IContext.Reload 32](#_Toc82006823)

[IContext.Resolve 32](#_Toc82006824)

[IContext.WithDeferred 32](#_Toc82006825)

[IContext extension methods 33](#_Toc82006826)

[CurrentUser 34](#_Toc82006827)

[NewGuid 34](#_Toc82006828)

[Now & Today 35](#_Toc82006829)

[RandomSeed 35](#_Toc82006830)

[WithInformUser & WithWarnUser 36](#_Toc82006831)

[Recognised Attributes 38](#_Toc82006832)

[Bounded 38](#_Toc82006833)

[CreateNew 38](#_Toc82006834)

[DefaultValue 39](#_Toc82006835)

[DescribedAs 40](#_Toc82006836)

[Disabled 40](#_Toc82006837)

[DisplayAsProperty 40](#_Toc82006838)

[Edit 41](#_Toc82006839)

[Hidden 41](#_Toc82006840)

[Mask 42](#_Toc82006841)

[MaxLength 42](#_Toc82006842)

[MemberOrder 42](#_Toc82006843)

[MinLength 43](#_Toc82006844)

[MultiLine 43](#_Toc82006845)

[Named 44](#_Toc82006846)

[Optionally 45](#_Toc82006847)

[PageSize 45](#_Toc82006848)

[Plural 45](#_Toc82006849)

[PresentationHint 46](#_Toc82006850)

[RegEx 46](#_Toc82006851)

[RenderEagerly 47](#_Toc82006852)

[TableView 48](#_Toc82006853)

[ValueRange 48](#_Toc82006854)

[Versioned 49](#_Toc82006855)

[ViewModel 49](#_Toc82006856)

[System configuration 50](#_Toc82006857)

[Persistor options 51](#_Toc82006858)

[Specifying the DbContext 51](#_Toc82006859)

[Configuring to work with Entity Framework 6 rather than Core 52](#_Toc82006860)

[RESTful API options 52](#_Toc82006861)

[Authentication & Authorization 55](#_Toc82006862)

[Authentication 55](#_Toc82006863)

[Authorization 56](#_Toc82006864)

[Troubleshooting 60](#_Toc82006865)

[Logging 60](#_Toc82006866)

[Writing safe LINQ queries 60](#_Toc82006867)

[Appendices 63](#_Toc82006868)

[Recognised Value Types 63](#_Toc82006869)

[Recognised Collection types 63](#_Toc82006870)

[Customizing the generic client 64](#_Toc82006871)

[I18N 65](#_Toc82006872)

# 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 main respects:

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

Expanding on the first of these points, Functional programming (FP) means building programs from pure, side-effect free, functions. 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.

While there are programming languages purpose-designed to support FP – such as Haskell or F# – it is also possible to implement pure FP in ‘mixed-paradigm’ programming languages such as C#.

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 ‘impure’ functions that handle the input-output. Impure functions may invoke pure functions, but not vice versa: if a pure function were to call a dirty function, then it becomes an impure function itself. The traditional objective is to keep the impure 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 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

The Naked Functions Template provides a very simple application example that might be considered as a start point for building a full-scale enterprise application: it contains only a single domain 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 domain types with millions of instances, 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 continuous 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 (referred to as the ‘Naked Framework’) and a common generic client, but they adopt radically different application programming approaches. Naked Objects is designed for pure object-oriented programming; Naked Functions is designed for pure functional programming.

You can find the templates:

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

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

* The Naked Functions Server solution (note that there is also a Naked Objects Server within the same main directory – we won’t be using that 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 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).

#### The Server solution

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.

#### The Client solution

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 instances 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 details of 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 instance and see what difference that makes. Then explore the other generic icons at the bottom of the screen.

Note that the student instance 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 instance or generate an instance of another domain type 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 multiple 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. (In a more substantial application, the application model might be spread across multiple model projects.) mportantly, it does *not* depend upon the Naked Functions framework. It does have a reference to the NakedFunctions.ProgrammingModel NuGet package, but this consists principally of attributes, plus a few interfaces including the very important IContext interface (see Using the IContext).

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

The actions rendered on the **Students** menu are defined as pure functions, on the static type Student\_MenuFunctions.For further information see Domain functions.

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. All domain types and domain functions need to be registered with the framework, but for the template solution this responsibility is delegated to a ModelConfig class in the Model project. For more information see Registering domain types and Registering domain functions.

# Domain types

An application written to work with Naked Functions will usually define many domain types. In a typical Naked Functions application, most of these domain types will be Persisted domain types. In addition, there may be Domain Interfaces, View Models, and Domain-specific enums, all of which may be displayed on the user interface. All four kinds of domain type need to be registered with the Naked Functions framework.

## Registering domain types

When you define *any* new domain type, it must be registered with the Naked Functions framework, as explained below.

**Important:** Entity Framework Core needs to be made aware, separately, of all*persisted types*. This may be done, minimally, by defining a property of type DbSet<T> on the DbContext. (Strictly speaking, it is only necessary to register any new types that cannot be reached by navigation from a type for which there is already a DbSet<T>. However, there is little disadvantage to creating a DbSet<T> property for each persisted type.)

Registration with Naked Functions is undertaken within the ConfigureServices method of the standard Startup class in the server project:

public void ConfigureServices(IServiceCollection services)

{

...

services.AddNakedFramework(frameworkOptions =>

{

...

frameworkOptions.AddNakedFunctions(appOptions =>

{

...

appOptions.DomainTypes = new Type[] { add domain types here};   
 ...

});

This list may be defined manually in code (typeof(Customer), typeof(Product) etc), or you may choose to use namespace and/or naming conventions with 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.DomainTypes = ModelConfig.DomainTypes();

which then finds all classes, interfaces, and enums, defined in the Template.Model.Types namespace:

public static Type[] DomainTypes() =>

PublicClassesInterfacesEnums.Where(t => t.Namespace == "Template.Model.Types" && !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;

## 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 following standard Entity Framework Core patterns.

Persisted domain types should be immutable classes, or as records.

#### Using immutable classes

public class Department  
{  
 public Department() {}

public Department(Department cloneFrom)  
 {  
 DepartmentID = cloneFrom.DepartmentID;  
 Name = cloneFrom.Name;  
 ModifiedDate = cloneFrom. ModifiedDate;  
 }

public short DepartmentID { get; init; }

public string Name { get; init; } = "";

public string GroupName { get; init; } = "";

public DateTime ModifiedDate { get; init; }

public override string ToString() => Name;  
}

Notes:

* Properties are specified with { get; init; } accessors, which ensures that the values may be set *only during construction.*
* Naked Functions does not require a domain type to have an explicit constructor. However, you may find it *convenient* to implement a constructor as shown above, that copies all properties (and collections if applicable) from a supplied cloneFrom instance of the same class. (You will then need to provide a zero-parameter constructor, too). This ‘cloning’ constructor may be used create a copy with all properties the same except for specified differences, for example:  
  => new Product(prod) {StockQty = newStockQty};

This ‘cloning’ constructor is not necessary when using records, because C# offers the with keyword for use with record types:  
=> prod with {StockQty = newStockQty};

#### Using records

Both Entity Framework Core and Naked Functions rely on reference equality for domain types. Since records implement value equality by default, you should override this by adding these two lines of code for each record type – illustrated here for Student:

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

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

**Warning:** At the time of writing, when running under .NET 6, Microsoft’s Entity Framework Core contains a significant bug that manifests if you are using *records* with inheritance (and also with ‘lazy loading’, which Naked Functions uses). Details of the bug may be found here: <https://github.com/dotnet/efcore/issues/26602> . Until Microsoft fixes this bug: if you wish to use inheritance in your model – speicfy those domain types involved in the inheritance hierarchy as immutable classes, rather than as records.

#### Attributes applyied at type level

Persisted domain types may be marked up – at *type* level – with any of these NakedFunctions attributes: Bounded, DescribedAs, Named, Plural, PresentationHint, RenderEagerly.

#### Title

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 persisted properties should define {get; init;} accessors.

All *navigable* properties (where the type of the property is a domain entity, or a collection of domain entities) should be virtual. Note that this requirement is not imposed by Naked Functions but by Entity Framework Core, to support lazy loading.

* Properties may be of any of the Recognised Value Types, or of any persisted domain type, or a collection of a persisted domain type.
* Properties of a single domain type may be marked up with any of these NakedFunctions attributes: DescribedAs, Hidden, Mask, MemberOrder, MultiLine, Named, PresentationHint, Versioned.
* Collection properties must be virtual, be of one of the Recognised Collection types, and initialised in code, for example:

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

* Collection properties may be marked up with these NakedFunctions attributes: DescribedAs, Hidden, MemberOrder, Named, PresentationHint, RenderEagerly, TableView.
* Properties or collections may be hidden based upon dynamic rules by defining a suitable Hide function.

## Domain Interfaces

A domain type may implement multiple interfaces, some of which will be technical in nature. An interface is said to be a domain interface if it plays a role in a domain function. For this reason, domain interfaces must be registered along with other domain types – see Registering domain types.

For example, an interface IHasEmailAddress might define a property representing the email address, and be implemented by multiple persistent domain types. This would then permit domain functions (SendEmail, ShowRecentMailReceived, perhaps) to be defined as an extension method on the type IHasEmailAddress, such that the functions would be rendered as actions in the Actions menu for each of the types implementing the interface.

## View Models

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 to display its details alongside the primary object
* Authorization may be used to control which properties may be seen by which users.

ViewModels are recommended only when it is necessary to construct a view built from more than persisted object – other than a simple list.

For example:

[ViewModel(typeof(CustomerDashboard\_Functions))]

public class CustomerDashboard {

[Hidden]

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

The ViewModel attribute must be given a static class that defines 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 given just these keys (thereby allowing the user to navigate back and forth between views). The signature of two 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 – see Registering domain types.

## Domain-specific enums

A domain-specific enum, if registered like other domain types (see Registering domain types), may be used as a property on a domain type, or as a parameter with a domain function.

Although the underlying type of an enum is a numeric type (and this is how an enum property of a persistent domain type will appear in the database) Naked Functions will, at the user interface, replace the integer value with a formatted version of the enum’s corresponding string value. For example, this enum:

public enum OrderStatus : byte {

InProcess = 1,

Approved = 2,

BackOrdered = 3,

Rejected = 4,

Shipped = 5,

Cancelled = 6

}

When used as a parameter in this function:

public static IQueryable<SalesOrderHeader> OrdersByStatus(IContext context, OrderStatus status) =>

context.Instances<SalesOrderHeader>().Where(x => x.StatusByte == (byte) status).OrderByDescending(obj => obj.OrderDate);

Will be rendered as a drop-down list on the generic client:

Graphical user interface, application, Teams

Description automatically generated

## How-to’s

#### How to implement a derived property or collection

See DisplayAsProperty.

# Domain functions

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 it *must*:

* be public, static, and defined on static class.
* depend only on values passed in as parameters to the function.
* be side-effect free – it must not modify any of the passed-in parameter values, nor make any change to the system. It 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 (see Using the 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 };

}

Domain functions are of four main types, all of which must be registered with the framework:

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

## Registering domain functions

Every function that is intended to be a user-action, or to provide supporting behaviour for a user action the functions 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 Startup class in the server project:

public void ConfigureServices(IServiceCollection services)

{

...

services.AddNakedFramework(frameworkOptions =>

{

...

frameworkOptions.AddNakedFunctions(appOptions =>

{

...

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

});

The Type[] 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 namespace and/or naming conventions with 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;

## 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 =>

{

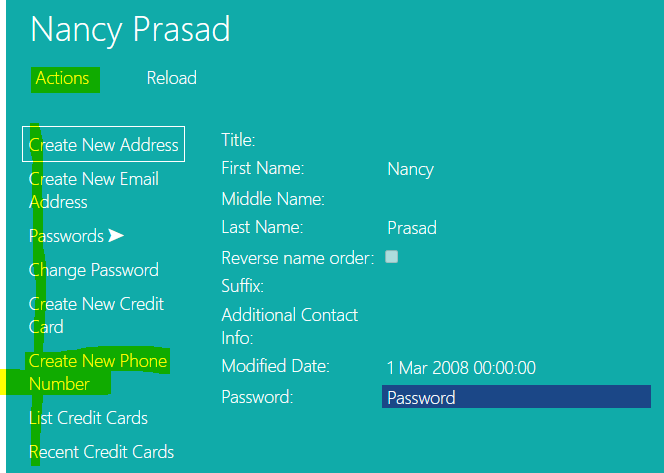
frameworkOptions.MainMenus = MenuHelper.GenerateMenus( array of types required );  
 ...

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 specified, 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

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.

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 parameter, 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). An 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

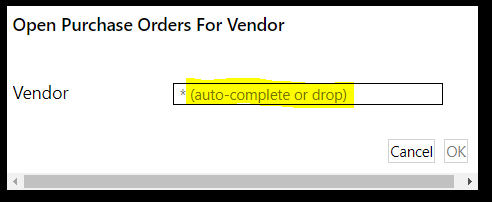
The user may specify the value for a domain type parameter by selecting from the choices (if specified – see Choices function), or by dragging and dropping a reference from elsewhere on the screen (including a second pane). A third option is to find the required instance by auto-complete, implemented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

## How to’s

### How to retrieve instances from the database

See IContext.Instances

### How to save new instances to the database

See IContext.WithNew.

### How to update existing instances in the database

See IContext.WithUpdated.

### How to access the current user

See CurrentUser.

### How to access the current date/time

See Now & Today.

### How to use random numbers

See RandomSeed.

### How to display feedback messages to the user

See WithInformUser & WithWarnUser.

### How to generate a GUID

See NewGuid.

### How to get hold of a registered service

See IContext.WithUpdated.

### How to implement a user action that has custom side-effects

See IContext.WithDeferred.

### How to write a function to edit a property in-line

See Edit attribute.

# Using the 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 instance passed in as a parameter. The returned result may comprise only new instances and/or unmodified existing instance accessed directly, or indirectly, via parameter 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.

## IContext.Instances

The IContext provides an Instances<T> method for accessing the instances of any persisted domain type as an IQueryable<T.

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;

## IContext.WithNew

The WithNew method captures an instance that needs to be persisted (for the first time) to the database. Note that:

* If you have set up the database to manage the IDs (this is the usual pattern) then the instance passed into WithNew will typically *not yet* have its ID field(s) set.
* WithNew does not modify the IContext – it returns a new instance of IContext, with the same state as the previous one, but with the new instance added to its internal list of instances to be persisted.
* WithNew does not make any call to the database or the system generally. The persistence is initiated by the Naked Functions framework *after the domain function has exited, and within a transaction.*

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 CreateNewWorkOrder 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 should 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 use the CreateNew attribute is optional, but it has a benefit at the user interface. See CreateNew.
* The WorkOrder displayed to the user when the framework has completed its work will have been retrieved back from the database after it has been persisted – complete with Id fields (assuming these are managed by the database).

## IContext.WithUpdated

The WithUpdated method captures an instance that needs to be updated in to the database. Note that:

* WithUpdated does not modify the IContext – it returns a new instance of IContext, with the same state as the previous one, but with the details of the updated instance added to its internal list of pending updates.
* WithUpdated does not make any call to the database or the system generally. The updating is initiated by the Naked Functions framework *after the domain function has exited, and within a transaction.*
* WithUpdated must be passed the original (unmodified) instance, and the updated version of this instance – which will be a new instance copied from the old one including, importantly, the unmodified Id field(s) and with the required difference(s) to other fields.

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:

* The AppendComment 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 target instance of SalesOrderHeader (the this parameter). Returning just an IContext indicates that after the framework has completed the updates, the user interface should display the updated version of the instance on which the user invoked the action. The tuple would be needed only if it was desired to display a new or different instance.

Here is an example of a slightly 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.GetService

GetService allows a function to access any service that has been registered in the systems configuration. It is used by several of the extension methods (see IContext extension methods) and thereby avoids creating dependencies on items not passed into the function (see ). But it may also be used to access custom system services.

However, care should be taken when calling functions on system services not to create side effects. See also IContext.WithDeferred.

## IContext.Reload

Reload will retrieve an updated version of a specified persistent domain object from the database.

*It is not typically necessary to call* Reload *in straightforward domain functions.* It is intended primarily for use with deferred functions. (See IContext.WithDeferred).

## IContext.Resolve

Because Naked Functions makes extensive use of lazy loading, the Resolve method provides a mechanism to ensure that an associated object has been loaded.

*It is not typically necessary to call* Resolve *in straightforward domain functions.* It is intended primarily for use with deferred functions. (See IContext.WithDeferred).

## IContext.WithDeferred

By far the majority of the side effects generated by user actions consist of writing to the database or updading the screen, and these are handled via generic mechanisms in the Naked Functions programming model that allow the corresponding domain functions to remain side-effect free.

Occasionally, it is necessary for user actions to generate other, custom, side effects. One example is where an action should result (typically in addition to the generic side effects) in sending an email. The following shows how this might be coded, making using also of an IEmailSender service:

public static IContext ConfirmShipment(this SalesOrderHeader ord, string trackingNo, IContext context) =>

context.WithUpdated(ord, ord with { Status = OrderStatus.Shipped, TrackingNo =   
 trackingNo }).WithDeferred(

c => {c.GetService<IEmailSender>().SendEmail(ord.Email, $"Shipped {ord.No}", "...");   
 return c;   
 }  
 );

WithDeferred takes in a Func<IContext, IContext>. This Func is invoked, by the Naked Functions framework, but only after the domain function (ConfirmShipment) has completed. Even though SendEmail will create side effects *when it is invoked*, ConfirmShipment remains a pure and side-effect free function.

WithDeferred may also be used to specify functionality that can only be invoked after a database transaction has completed – for example because the function might depend on being able to read and use the database-generated Id on a new object. In the following example, WithDeferred is used to call the Recalculated function on a sales order, *after* the database has been updated with a change to the order:

public static IContext ChangeQuantity(this SalesOrderDetail detail, short newQuantity, IContext context)

{

var sop = Product\_Functions.BestSpecialOfferProduct(detail.Product, newQuantity, context);

var detail2 = detail with

{

OrderQty = newQuantity,

SpecialOfferProduct = sop,

UnitPrice = detail.Product.ListPrice,

UnitPriceDiscount = sop.SpecialOffer.DiscountPct,

ModifiedDate = context.Now()

};

return context.WithUpdated(detail, detail2).WithDeferred(

c => {

var soh = c.Resolve(detail.SalesOrderHeader);

return c.WithUpdated(soh, soh.Recalculated(c));

}

);

}

Note the use of the Resolve method also (see IContext.Resolve).

## IContext extension methods

The following functions are implemented as extension methods to the IContext. Note:

* The implementations of these methods all delegate to system services, via the IContext.GetService method.
* All these extension methods are side-effect free; some return a new instance of IContext.

### CurrentUser

Provides access to the IPrincipal containing details of the current user (if the application requires a logon). For example:

public static IContext AddProductReview(this Product p,

DateTime dateOfReview,

int rating,

string comments,

IContext context) =>

context.WithNew( new ProductReview

{

Product = p,

ReviewerName = context.CurrentUser().Identity.Name,

ReviewDate = dateOfReview,

...

});

}

The implementation of CurrentUser delegates to a method on the IPrincipalProvider service:

public static IPrincipal CurrentUser(this IContext context)

=> context.GetService<IPrincipalProvider>().CurrentUser;

A default implementation of IPrincipalProvider is provided by Naked Functions, but you may create and register your own implementation of this service, for example to facilitate testing.

### NewGuid

This is equivalent to calling System.Guid.NewGuid(), but avoids creating an illegal system dependency within a function:

public static SalesOrderDetail CreateNewDetail(this SalesOrderHeader soh, Product product, short quantity, IContext context)

{

var specialOfferProduct = Product\_Functions.BestSpecialOfferProduct(product, quantity, context);

return new SalesOrderDetail()

{

SalesOrderHeader = soh,

OrderQty = quantity,

SpecialOfferProduct = specialOfferProduct,

UnitPrice = product.ListPrice,

UnitPriceDiscount = specialOfferProduct.SpecialOffer.DiscountPct,

rowguid = context.NewGuid(),

ModifiedDate = context.Now()

};

}

The implementation of NewGuid delegates to a method on the IGuidGenerator service:

public static Guid NewGuid(this IContext context)

=> context.GetService<IGuidGenerator>().NewGuid();

A default implementation of IGuidGenerator is provided by Naked Functions, but you may create and register your own implementation of this service, for example to facilitate testing.

### Now & Today

This is equivalent to calling System.DateTime.Now, but avoids creating an illegal system dependency within a function:

public static IContext AddProductReview(this Product p,

DateTime dateOfReview,

int rating,

string comments,

IContext context) =>

context.WithNew( new ProductReview

{

...

Comments = comments,

ModifiedDate = context.Now()

});

}

The implementation of Now delegates to a method on the IClock service:

public static DateTime Now(this IContext context)

=> context.GetService<IClock>().Now();

Today works in a similar fashion to Now, but equivalent to System.DateTime.Today.

A default implementation of IClock is provided by Naked Functions, but you may create and register your own implementation of this service, for example to facilitate testing.

### RandomSeed

Working with random numbers creates specific challenges in FP for several reasons:

* Pure functions must be deterministic: the returned value must depend solely and deterministically on the values of the parameter(s) passed in.
* Passing in an instance of System.Random would not work, because calling Next on that instance would *modify the state of the instance*.

RandomSeed provides a solution, and follows a common pattern used for random numbers in FP. The passed in context will have a random seed (of type NakedFunctions.IRandom) already provided by the Naked Functions framework. Methods on IRandom will transform this into an integer in a specified range. The following simple example retrieves a random instance of an Individual from the database:

public static Person RandomPerson(IContext context)

{

var persons = context.Instances<Person>().Where(t => t.PersonID != null).OrderBy(t => "");

var random = context.RandomSeed().ValueInRange(persons.Count());

return persons.Skip(random).FirstOrDefault();

}

Calling the Next() function in an IRandom will return a new pseudo-random IRandom, generated deterministically from the previous one. The following more complex example uses an internal *recursive* function to create a list of randomly-selected persons:

public static IList<Person> RandomPersons(int numberRequired, IContext context) =>

RandomPersons(numberRequired, context.Instances<Person>().OrderBy(p => ""),   
 context.RandomSeed()).ToList();

// Recursive function

internal static ImmutableList<Person> RandomPersons(

int num, IOrderedQueryable<Person> source, IRandom random) =>

num < 1 ? ImmutableList<Person>.Empty :

ImmutableList.Create(RandomSelection(source, random))  
 .AddRange(RandomPersons(num - 1, source, random.Next()));  
  
// Helper function

internal static Person RandomSelection(IOrderedQueryable<Person> source, IRandom random) =>

source.Skip(random.ValueInRange(source.Count())).First();

The implementation of RandomSeed delegates to a method on the IRandomSeedGenerator service:

public static IRandom RandomSeed(this IContext context)

=> context.GetService<IRandomSeedGenerator>().Random;

A default implementation of IRandomSeedGenerator is provided by Naked Functions, but you may create and register your own implementation of this service, for example to facilitate testing.

### WithInformUser & WithWarnUser

Sometimes it is useful to provide the user with brief feedback messages. The WithInformUser and WithWarnUser extension methods do this. For example:

public static IContext AddToShoppingCart(Product product, IContext context) {

string id = GetShoppingCartIDForUser(context);

var newItem = new ShoppingCartItem() with { ShoppingCartID = id, Product = product, Quantity = 1, DateCreated = context.Now()};

return context.WithNew(newItem).WithInformUser($"1 x {product} added to Cart");

On the generic client, this message will appear in the screen footer, until another action is performed. (The only distinction between an ‘inform’ and a ‘warn’ on the generic client, is that the latter is rendered in red – but they could be treated differently by altering the .css, or by further client customisation).

The implementation of WithInformUser and WithWarnUser delegate to methods on the IAlert service:

public static IContext WithWarnUser(this IContext context, string message)

=> context.WithDeferred(c => {

c.GetService<IAlert>().WarnUser(message);

return c;

});

public static IContext WithInformUser(this IContext context, string message)

=> context.WithDeferred(c => {

c.GetService<IAlert>().InformUser(message);

return c;

})

Note that in both cases, the function being delegated to is not called directly, but is registered as a deferred function (see IContext.WithDeferred).

A default implementation of IAlert is provided by Naked Functions, but you may create and register your own implementation of this service, for example to facilitate testing.

# Recognised Attributes

These attributes are all defined with the NakedFunctions namespace, and are installed as part of the NakedFunctions.ProgrammingModel NuGet package.

### 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 class 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) description, 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 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/ collapsible 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 minimum 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 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 RESTful API for custom use. For example:

[MultiLine(10)]

public 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 class 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 class 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.
* 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 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 IContext.WithDeferred).

### ViewModel

Applied to a domain type, specifies that this is a view model, not a persisted type. See View Models.

# System configuration

The Naked Functions framework is configured as a set of services registered in within an ASP.NET Core Web project, as shown in the following code from the Startup.cs file (taken from the Naked Functions Server template project):

public void ConfigureServices(IServiceCollection services)

{

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>

{

options.Authority = $"https://{Configuration["Auth0:Domain"]}/";

options.Audience = Configuration["Auth0:Audience"];

options.TokenValidationParameters.NameClaimType = "http://schemas.xmlsoap.org/ws/2005/05/identity/claims/emailaddress";

});

services.AddControllers()

.AddNewtonsoftJson(options => options.SerializerSettings.DateTimeZoneHandling = DateTimeZoneHandling.Utc);

services.AddMvc(options => options.EnableEndpointRouting = false);

services.AddHttpContextAccessor();

services.AddNakedFramework(frameworkOptions =>

{

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

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

frameworkOptions.AddNakedFunctions(appOptions =>

{

appOptions.FunctionalTypes = ModelConfig.DomainTypes();

appOptions.Functions = ModelConfig.DomainFunctions();

});

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

});

services.AddCors(corsOptions =>

{

corsOptions.AddPolicy(MyAllowSpecificOrigins, policyBuilder =>

{

policyBuilder

.WithOrigins("http://localhost:5001")

.AllowAnyHeader()

.WithExposedHeaders("Warning", "ETag", "Set-Cookie")

.AllowAnyMethod()

.AllowCredentials();

});

});

}

The key section is highlighted, but note that the ConfigureServices also:

* Specifies the authentication mechanism (in this case Auth0). For more information, see Authentication.
* Specifies the CORS policies. Note specifically that provision has been made for allowing requests originating from <http://localhost:5001> – which is the URL to which the Template Client project is deployed by default. If you deploy the client to another URL or port, you will need to change (or add to) the CORS configuration.

Turning back to the highlighted section, services.AddNakedFramework sets up the ‘Naked Framework’, which is the common functionality used by both the Naked Functions and Naked Objects application frameworks. The following framework options are specified at this level – because they work the same way for both application frameworks:

* Specification of main menus:   
  frameworkOptions.MainMenus = MenuHelper.GenerateMenus(...);   
  For further information see Main menu functions.
* Specification of the persistor mechanism: frameworkOptions.AddEFCorePersistor(peristorOptions => { peristorOptions.ContextCreators = new[] {...}; });  
  For further information see Persistor options.
* Specification of the RESTful API. This single line: frameworkOptions.AddRestfulObjects(\_ => { });  
  is sufficient to ensure that the server creates a complete RESTful API that reflects the full capabilities of your domain model. However, there are further options that may be specified within this. For further information, see RESTful API options.
* Specification of authorization mechanism – not shown here, but for further information see Registering authorizers.

## Persistor options

By default, the Template.Server project is configured to work with Entity Framework Core – see Configuring to with Entity Framework 6 rather than Core.

### Specifying the DbContext

The framework’s persistor must be advised how to create a DbContext corresponding to the model whenever needed.

This is done by registering a function that will create the DbContext. For example:

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

In this example, responsibility is delegated to the ModelConfig, where the function to create a DbContext when needed is defined thus:

public static Func<IConfiguration, DbContext> EFCoreDbContextCreator =>

c => {

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

db.Create();

return db;

};

Note that, because the peristorOptions.ContextCreators takes an array (of type Func<IConfiguration, DbContext> ), it is straightforward to split the model across multiple DbContexts if this is required.

### Configuring to work with Entity Framework 6 rather than Core

Naked Functions may 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.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 function to create a DbContext *that follows the EF6 patterns*).

## RESTful API options

The Naked Functions framework automatically generates a RESTful API that reflects the full capabilities of the domain model.

If you are using the generic client it is not necessary to understand anything about the RESTful API, but if you were to write your own client *from scratch*, or wanted to interface another system via the RESTful API, then comprehensive documentation may be found in the **Naked Framework Restful API specification**, which may be downloaded from: <https://github.com/NakedObjectsGroup/NakedObjectsFramework/tree/master/Documentation>

Some options may be

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

Functions that return an IContext are assumed to be making persistent changes, so the corresponding resource on the RESTful API must be accessed via an Http POST method.

Functions that do not return an IContext, and resources for accessing instances or their members, do not make persistent changes, so the corresponding resources on the RESTful API are accessed via an Http GET method.

#### Cache settings

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

**Naked Framework client-managed caching** functions independently of the browser’s generic caching mechanism. Thiscache 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 domain 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>();

# Authentication & Authorization

Authentication refers to the ability to identify the user and control their access to the system at the top level. Authorization refers to the ability to control what an *authenticated* user can see and do within an application, based upon their identity, the role(s) assigned to them, and/or other credentials or 'claims'. See Authentication.

Naked Functions supports 'fine-grained authorization', meaning that it is possible to specify whether a user may invoke individual main menu actions, view specific domain types or even specific instances within a type, view individual properties or collections within a type, o invoke individual actions on a type. See Authorization.

Note that if access to any resource is not authorized, the resource will not appear in the user interface, nor in the links provided within the RESTful API from the server. Even if a user attempting to bypass the client were to work out the server URL for that resource and call it with correct arguments, the server would still reject the request.

## Authentication

The [Naked Functions Template solution](#_Running_NOF9_with) is set up to provide a straightforward approach to user authentication, where the application requires it. It makes use of the [Auth0](https://manage.auth0.com/) login service, which, in turn, allows you to log in using your Google, Facebook, or other recognised account. To use this capability in your 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. On the Auth0 [website](https://auth0.com/), first create an account if you do not already have one, 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.
   1. The *Domain*, which you will have chosen when you set up your account e.g. FooBar.eu.auth0.com
   2. The *Client ID* (a string of letters and numbers)
   3. 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.
6. 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": "FooBar.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 – see Authorization.

## Authorization

Authorization is implemented by writing one or more ‘authorizer’ classes, specifically:

* [*Mandatory*]A *default* ‘type authorizer’ which will manage the authorization to view any domain types and potentially access to individual properties, and the actions on the Actions menu for a type. See The default type authorizer.
* [*Mandatory*]A single ‘main menu authorizer’ which will manage the authorization of all main menu actions in one place. See The main menu authorizer.
* [*Optional*] Further type authorizers specifying authorization for individual domain types (implementing ITypeAuthorizer<DomainType>) and/or for multiple types within a specified namespace (implementing INamespaceAuthorizer). See Type authorizers and Namespace authorizers.

The authorizers must all be registered with the framework – see Registering authorizers.

When operating:

* When the framework is asked to provide main menus (usually at start-up, as the SPA client will cache these) the main menu authorizer’s IsVisible method will be called for each action on each main menu:
* Before the framework displays any instance of a domain type, it will find the *most specific authorizer* that has been registered that fits the type, and then call its IsVisible method for each member to be displayed: a type authorizer for a single domain type will *usually* be more specific than a namespace authorizer. If no specific matching type authorizer or namespace authorizer is found, it will delegate responsibility to the *default* type authorizer which, by definition, can handle all domain types.

#### Registering authorizers

All the authorizers must be registered with the framework using an IAuthorizationConfiguration. This might be returned by a custom function, for example:

public static IAuthorizationConfiguration MyAuthConfig() {

var config = new AuthorizationConfiguration<MyDefaultTypeAuthorizer, MyMainMenuAuthorizer>();

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

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

config.AddTypeAuthorizer<Bar, MyBarAuthorizer>();

return config;

}

Notes:

* The NakedFunctions.AuthorizationConfiguration class requires the mandatory default type authorizer and the mandatory main menu authorizer to be specified as generic parameters.
* Optional type authorizers and/or namespace authorizers may then be added to the class. The order in which they are added, does not matter.

The implementation of IAuthorizationConfiguration must then be provided to the framework as shown here:

services.AddNakedFramework(frameworkOptions =>

{

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

frameworkOptions.AddNakedFunctions(appOptions =>

{

appOptions.FunctionalTypes = ModelConfig.DomainTypes();

appOptions.Functions = ModelConfig.DomainFunctions();

});

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

frameworkOptions.AuthorizationConfiguration = MyAuthConfig();

});

Notes:

* AuthorizationConfiguration is a property in the frameworkOptions on the NakedFramework service, not in the appOptions of the NakedFunctions service. (NakedFramework defines functionality that is common between the NakedFunctions and the NakedObjects frameworks). Looking at the code above, you can see that NakedFunctions is itself registered in the frameworkOptions.

#### The default type authorizer

The default type authorizer must implement NakedFunctions.Security.ITypeAuthorizer<object> and acts as the ‘last resort’ for authorization relating to display of a domain type and its associated properties and/or actions. It must implement this method:

public bool IsVisible(object target, string memberName, IContext context)

Unless a more specific type authorizer or namespace authorizer provides a match, the method above will be called each time a domain type is displayed, and for each member (property or action).

One approach favoured by some developers is to make this default method always return false, and then gradually make model capabilities available by writing and registering more specific authorizes. See Type authorizers and Namespace authorizers.

Notes:

* The IContext (in this and all the other authorizer implementations) may be used to look up other relevant instances from the database.
* The IsVisible method deliberately does offer the option to return an IContext, because calling this method should not involve any updates to the database or the screen.

#### The main menu authorizer

To manage authorization of main menu actions you need to write an implementation of NakedFunctions.Security,IMainMenuAuthorizer. This interface defines a single method:

public bool IsVisible(string target, string memberName, IContext context)

which will be called during the start-up process, for each of the possible registered main menu functions, to determine whether the function should be displayed for the currently logged-on user (if any). When called, target will be the fully qualified name of the static type on which the menu function is defined, and memberName will be the unformatted version of the name of the function.

#### Type authorizers

A type authorizer must implement NakedFunctions.Security.ITypeAuthorizer<T> where T is a domain type – which may be abstract or concrete. It must then provide:

public bool IsVisible(T target, string memberName, IContext context)

#### Namespace authorizers

A namespace authorizer must implement NakedFunctions.Security.ITypeAuthorizer<T> where T is a domain type – which may be abstract or concrete. The namespace authorizer implements the same method as on the default type authorizer i.e.:

public bool IsVisible(object target, string memberName, IContext context)

but – because of how it is registered – the programmer may assume that this method is called only when the target object is defined within the namespace for which it was registered. See Namespace authorizers.

# 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 Functions 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 nakedfunctions.log which will be located within the bin directory of the startup project.

If suitably configured, the Naked Functions framework can record a large range of useful information, including warnings about functions or types that do not match the expected patterns. These often turn out to the cause when the generated user interface does not match your expectations from the domain code.

#### Logging within domain code

It is also possible to use the same logging infrastructure to log events within your domain code. You can access the logging service with IContext.GetService.

## Writing safe LINQ queries

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

Don't write:

public static IQueryable<Product> FindProducts(ProductCategory category, IContext context) => context.Instances<Product>().Where(x => x.Category == category);

Write:

Public static IQueryable<Product> FindProducts(ProductCategory category, IContext context) => context.Instances<Product>().Where(x => x.Category.Id == category.Id);

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

Don't write:

public static IQueryable<Product> ListDiscontinuedProducts(IContext context) =>

context.Instances<Product>().Where(x => x.IsDiscontinued());

Write:

public static IQueryable<Product> ListDiscontinuedProducts(IContext context) =>

context.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 instances passed into the query; pass in any such required indirect references as variables in their own right

Don't write:

public static IQueryable<Order> OrdersSentToOtherAddress(Customer cust, IContext context) => context.Instances<Order>().Where(x => x.SentTo.Id != cust.BillingAddress.Id);

Write:

public static IQueryable<Order> OrdersSentToOtherAddress (Customer cust, IContext context) {

Address billingId = cust.BillingAddress.Id;

return Instances<Order>().Where(x => x.SentTo.Id != billingId);

}

#### Don’t use context.Instances<T>() more than once inside a query; define a separate IQueryable<T> outside the query

Don't write:

var q = from p in context.Instances<Product>()

from c in context.Instances<Customer>()

where ...

Write:

var customers = context.Instances<Customer>();

var q = from p in context.Instances<Product>()

from c in customers

where …

Or, for greater clarity:

var customers = context.Instances<Customer>();

var products = context.Instances<Product>();

var q = from p in products

from c in customers

where …

# Appendices

## Recognised Value Types

Naked Functions recognises the following .NET types as value types:

* 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 Functions in two different contexts:

1. As returned by a function that is registered as a user action
2. As a property on a domain type, representing a multiple association

These are defined below.

#### Collections returned by a function

In this context Naked Function recognises as a collection any implementation of IEnumerable<T> where T is a domain entity type (class, record, 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 Functions 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 Functions.

## Customizing the generic client

The generic client may be customised to any extent required, though the developer is expected to be fully conversant with the Angular framework.

Customising the *styling* of the generic client is straightforward and recommended. Addition of custom type views and/or behaviour involves more work – as with any Angular client. We recommend that this be deferred as long as possible in the development process because:

* Prototyping with the purely generic client (customised only at the CSS level) forces domain stakeholders and developers to focus on the core domain functionality, not the presentation.
* This results in faster prototyping and far more agility
* It also makes it easier to realise the full benefits of functional programming.
* Many teams have reported that though their starting intention was to develop a fully custom user interface, the business stakeholders reported that they found the generic client to be very effective, and opted finally just to do some simple restyling using CSS.

There is a separate document available to help you get started for this entitled ‘**The Naked Objects Client - Configuration and Customisation**’. It may be downloaded from here:

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

## I18N