**Naked Functions**

**Application Developer Manual**

**[under construction]**

Contents

[Introduction 4](#_Toc81381624)

[What is Naked Functions 4](#_Toc81381625)

[Starting from the Naked Functions Template solution 4](#_Toc81381626)

[Server solution 4](#_Toc81381627)

[Client solution 5](#_Toc81381628)

[Writing your own applications 5](#_Toc81381629)

[Defining domain types 6](#_Toc81381630)

[Properties 6](#_Toc81381631)

[Attributes for use on properties 6](#_Toc81381632)

[Collections 6](#_Toc81381633)

[Attributes for use on collections 6](#_Toc81381634)

[Overriding default methods common to all .NET types 7](#_Toc81381635)

[Overide the ToString method to define a title for an instance 7](#_Toc81381636)

[[Temp] override the GetHashCode and Equals methods 7](#_Toc81381637)

[Class level attributes 7](#_Toc81381638)

[Defining domain functions 8](#_Toc81381639)

[Attributes for use on functions 8](#_Toc81381640)

[Attributes for use on parameters 8](#_Toc81381641)

[System services & configuration 9](#_Toc81381642)

[Authentication 9](#_Toc81381643)

[Authorization 9](#_Toc81381644)

[Auditing 9](#_Toc81381645)

[Profiling 9](#_Toc81381646)

[I18N 9](#_Toc81381647)

[Appendices 10](#_Toc81381648)

[Attributes – summary 10](#_Toc81381649)

# Introduction

## What is Naked Functions?

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

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

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

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

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

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

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

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

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

## Starting from the Naked Functions Template solution

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

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

You can find the template solutions here:

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

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

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

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

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



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

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

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

Graphical user interface

Description automatically generated with medium confidence

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

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

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

The server solution comprises two projects.

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

This model defines the Student type:

public record Student

{

[Hidden]

public virtual int Id { get; init; }

public virtual string FullName { get; init; }

public override string ToString() => FullName;

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

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

}

Notes:

* Student is defined as a record. It may be defined as class, but, as of C# 9 records are easier to work with when adopting the FP approach. (Future versions of C# might drop the distinction between records and classes). Either way, types used with Naked Functions should be *immutable* – all properties are defined with { get; init; } accessors, so that the properties may only be set when the instance is being constructed. (We’ll see later that updating is done by creating a new instance as a copy of the existing one with the required differences specified.)
* The Id property (one is required for all persisted types, by EntityFrameworkCore) has been marked up with the (optional) Hidden attribute indicating – in this case – that a property is never displayed to a user.
* The ToString method has been overridden. This is used by Naked Functions to define the title for the record when displayed to the user – which, in this case, is just a copy of the FullName property, but could be a compound of several fields.
* The GetHashCode and Equals methods, found on all C# types, have been overridden using standard boilerplate. This is not required by Naked Functions, but, is currently required for working with Entity Framework Core using the ‘lazy loading’ approach. For the same reason, all properties must be defined as virtual (any any collections, not shown here, initialized in the code).

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

[Named("Students")]

public static class Student\_MenuFunctions

{

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

{

var s = new Student { FullName = fullName };

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

}

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

context.Instances<Student>();

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

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

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

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

}

Notes:

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

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

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

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

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

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

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

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

}

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

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

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

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

public void ConfigureServices(IServiceCollection services)

{

...

services.AddNakedFramework(frameworkOptions =>

{

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

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

frameworkOptions.AddNakedFunctions(appOptions =>

{

appOptions.FunctionalTypes = ModelConfig.Types();

appOptions.Functions = ModelConfig.Functions();

});

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

});

ModelConfig is defined and maintaned in the model project.

public static class ModelConfig

{

public static Type[] Types() =>

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

public static Type[] Functions() =>

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

public static Type[] MainMenus() =>

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

public static Func<IConfiguration, DbContext> EFCoreDbContextCreator =>

c => {

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

db.Create();

return db;

};  
 ...

}

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

#### Server solution

Sets up the system using standard Microsoft patterns

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

Link to system service configuration section

### Client solution

The only things you *need* to do.

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

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

## Writing your own applications

Define domain types

Define domain functions

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

Register domain types with Naked Functions, manually or reflectively

Register all domain functions with NakedFunctions, manually or reflectively

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

# Defining domain types

Types should be immutable. Can be immutable classes, but, for C#9 recommendation is to use records, because it permits the use of ‘with’. Later versions of C# are likely to eliminate remaining distinctions.

## Properties

Can be of any domain type, or fixed list of value types

use get; init;

All properties must be virtual (on persistent objects

### Attributes for use on properties

DescribedAs

Hidden

Mask

MemberOrder

MultiLine

Named

Versioned (ref also the

## Collections

### Attributes for use on collections

TableView

RenderEagerly

## Overriding default methods common to all .NET types

### Overide the ToString method to define a title for an instance

Building titles using string interpolation

### [Temp] override the GetHashCode and Equals methods

## Class level attributes

Bounded

DescribedAs TODO: review

Named

Plural

# Defining domain functions

### Attributes for use on functions

CreateNew

DescribedAs

Disabled

DisplayAsProperty

Edit

MemberOrder

Named

### Attributes for use on parameters

DefaultValue

DescribedAs

Disabled

Mask

MaxLength

MinLength

MultiLine

Named

# System services & configuration

General introduction to configuring the

## Authentication

## Authorization

## Auditing

## Profiling

## I18N

# Appendices

## Attributes – summary

Table, derived from existing spreadsheet.