**A picture containing night sky

Description automatically generated**

**The Naked Objects SPA Client:**

**Configuration and Customisation**

**Updated: 31st August 2021**

Contents

[Using the generic client 2](#_Toc81913319)

[Configuring the generic client 3](#_Toc81913320)

[Configuring colours for objects 3](#_Toc81913321)

[Configuring masks 5](#_Toc81913322)

[Shortening Urls 6](#_Toc81913323)

[Obfuscating or encrypting sensitive data in client URL query strings 7](#_Toc81913324)

[Other configurable client elements 8](#_Toc81913325)

[Customising the generic client using Angular 9](#_Toc81913326)

[Adding custom views 10](#_Toc81913327)

# Using the generic client

The default user interface for NOF is known as Gemini. It has modern or ‘flat’ styling, and makes optional use of coloured backgrounds to help distinguish the most important domain types – see [Configuring colours for objects in the Gemini user interface](#_Configuring_colours_for).

We strongly recommend that you run Gemini in full screen mode, since all the browser functions that you need are provided as icons at the bottom of the screen.

The Gemini name reflects the fact that it supports ‘twin panes’ – the screen may be split into two panes, allowing you to view and use two different instances or lists of instances at once.

Gemini has been designed to use familiar modes of interaction and should require no up-front training – just explore clicking on the obvious icons, buttons and links. However, two things may not be obvious:

* When clicking on an active area, such as an action, icon, or link (shown as a block of contrasting colour or tint), you have *in many cases* the option to click with the left or right mouse-buttons. The left (standard) button will open the next view in the same pane; the right mouse-button will open the next view in the *other* pane, splitting the screen if it is not already split.
* Gemini supports a limited form drag and drop. If you have a field in a dialog box, or an object in Edit view that requires a particular type of domain object, then you can drag and drop an instance of the correct type from the other pane. You can either drag a ‘link’ (small coloured bar) from a list, or if you are looking at a view of a single object, you can drag that object by its title.

# Configuring the generic client

Various aspects of the generic user interface may be configured, without required the more complex process of [re-building the client from source](#_Re-building_the_NOF9). These configurations are all specified within the file config.json.

The only mandatory element in config.json is the "appPath", which specifies the Url of the server (i.e. the RESTful API) All others elements - as described below - are optional

## Configuring colours for objects

By default, the client will assign a random colour to each domain object type (though the same colour will be used, consistently, for a given type – because the selection is based on a hash of the fully-qualified type name).

In config.json the randomMaxIndex property specifies the number of *different* colours that will be assigned. The actual colours are specified, by number, in themes.css, typically with different tints for the object background colour, and the link background colour – so that a link to an object of the same type will still show up as a distinct visual element. Text colour can also be specified in each case, though the default version specifies a single text colour. For example:

/\*0. Black \*/

    --link-color0: #404040;

    --object-color0: #000000;

    --link-text-color0: var(--contrast-text-color);

    --object-text-color0: var(--contrast-text-color);

/\*1. Blue\*/

    --link-color1: #1b4787;

    --object-color1: #2b5797;

    --link-text-color1: var(--contrast-text-color);

    --object-text-color1: var(--contrast-text-color);

Setting randomMaxIndex to 0 will disable random colour allocation and all objects will then be rendered in the default colour, which is specified by the default property.

The colours for individual object types may determined by the rules specified in config.json. The following shows an example of those rules:

"colors": {

"typeMap": {

"AdventureWorksModel.Customer": 1,

"AdventureWorksModel.Product": 4,

"AdventureWorksModel.Employee": 5,

"AdventureWorksModel.SalesPerson": 6,

"AdventureWorksModel.SpecialOffer": 7,

"AdventureWorksModel.Vendor": 9,

"AdventureWorksModel.WorkOrder": 12

},

"regexArray": [

{

"regex": ".\*SalesOrder.\*",

"color": 2

},

{

"regex": ".\*PurchaseOrder.\*",

"color": 10

}

],

"subtypeMap": {

"AdventureWorksModel.BusinessEntity": 8

},

"default": 0

},

There are three forms of rule in the example code above. The first section (the typeMap) specifies colours for individual object tyles (each specified by its fully qualified name). All colours are specified as numbers. The numbers are translated into actual colours by the .css (described below).

The next section (regexArray) uses regular expressions to specify the colour for any object type whose fully qualified name matches a given regex pattern. For example, the following code would specify colour number 2 for both SalesOrderHeader and SalesOrderDetail.

"regex": ".\*SalesOrder.\*",

"color": 2

The subtypeMap section will specify the colour for any sub-type of a given type (which may be a class or an interface). For example, the following code specifies colour number 8 for all sub-types of BusinessEntity:

"AdventureWorksModel.BusinessEntity": 8

Note: matching sub-types is a relatively expensive mechanism in terms of round-trips to the server (though the results are cached), so it should be used sparingly.

The rules are applied in the order shown above, and the colour for a given object type will be determined by the first rule that matches the type.

The colour numbers are interpreted in the styles.css file, for example:

.object-color0 {

background-color: #000000; /\*Black\*/

}

.link-color0 {

background-color: #303030 /\*#525252\*/ ; /\*Dark grey\*/

}

.object-color1 {

background-color: #2b5797; /\*Blue\*/

}

.link-color1 {

background-color: #1b4787; /\*Slightly-lighter blue\*/

}

For each colour number there are two variants: the .object-color and the .link-color. The first is used as the background for a view of that type of object, and the second it used as the background for a link to an object. The recommended approach is to keep these two colours similar, but with a slight difference in brightness. This is so that a link will always show up against a background that has the same colour number. In order to change any of the .css you will need to [re-build the client form source](#_Running_NOF9_with_2).

## Configuring masks

The client source code defines a default format for each of the value types. Masks, for formatting value strings, may be specified in the domain model, using the [Mask](#d0e5093) attribute. However, the interpretation of these masks is the responsibility of the client and this may be specified in config.js as shown in the following example code:

"masks": {

"currencyMasks": {

"C": {

"format": "decimal",

"symbol": "GBP",

"digits": "1.2-2"

},

"c": {

"format": "decimal",

"symbol": "GBP",

"digits": "1.2-2"

}

},

"dateMasks?": {

"d": {

"format": "date-time",

"mask": "D MMM YYYY",

"tz": "+0000"

},

"D": {

"format": "date",

"mask": "D MMM YYYY hh:mm:ss"

},

"T": {

"format": "time",

"mask": "HH:mm",

"tz": "+0000"

}

}

},

For example, [Mask("d")] will present only the date part of a DateTime value, and [Mask("c")] applied to a Decimal property will render it as a GBP currency value to two decimal places.

**Note:**  The specification of the value types in the code above (for example: "decimal", "date-time") are the types as defined in the [Naked Objects RESTful API](https://github.com/NakedObjectsGroup/NakedObjectsFramework/blob/master/Documentation/RestfulAPI.doc).

## Shortening Urls

The Urls used to navigate within the Spa client may be shortened using the urlShortCuts element in config.json. (Note that these Urls are not sent to the server – those Urls all follow the format specified in the RESTful Api). For example, the following code:

"urlShortCuts": [ "http://nakedobjectsrodemo.azurewebsites.net", "AdventureWorksModel" ],

Specifies that wherever the strings <http://nakedobjectsrodemo.azurewebsites.net> or AdventureWorksModel appear within the client Url, these will be compressed, for example into the form \_\_\_0, \_\_\_1, etc, where \_\_\_ is a specified shortCutMarker.

## Obfuscating or encrypting sensitive data in client URL query strings

For some applications and environments, security concerns extend to the possibility of sensitive information being visible in query-string part of the URLs used by the browser, and hence the possibility of this information being cached *by the browser*. (These ‘client URLs’ are not the same as the URLs on the server being accessed, behind the scenes.)

If this is a concern, then data elements in the client-URLs may be obfuscated or encrypted.

To do this you will need to provide an implementation of the obfuscate service defined here: nakedobjectsspa\services\src\obfuscate.service.ts

The framework comes with a simple example implementation here:

import { Injectable } from '@angular/core';

@Injectable()

export class Base64ObfuscateService {

public obfuscate(s: string) {

return s ? btoa(s) : s;

}

public deobfuscate(s: string) {

return s ? atob(s) : s;

}

}

To use this service, edit the file nakedobjectsspa\src\app\app.module.ts and uncomment the two highlighted lines shown below,

// import { Base64ObfuscateService } from './base64obfuscate.service';

@NgModule({

declarations: [

AppComponent,

],

imports: [

...

],

providers: [

// {provide : ObfuscateService, useClass: Base64ObfuscateService}

],

bootstrap: [AppComponent]

})

export class AppModule { }

and re-build the client app. You will now see that object Ids and any dialog input parameters, shown in the query string on the client URL have been obfuscated.

Note that this example service uses base 64 conversion, making the data hard for a human to read – but it is not a proper form of encryption. You should replace this example with your own service that implements the same two simple methods, using a form of encryption appropriate to your technical environment and security needs, and register your service in place of the Base64ObfuscateService in app.module.ts.

## Other configurable client elements

Several other aspects of the client may optionally be configured within config.json. These are not yet fully documented, but the list of recognised optional elements in config.json is defined (in the source code) by the IAppConfig interface. The following is an excerpt of that interface, showing properties that have not been mentioned above:

export interface IAppConfig {

appPath: string;

applicationName : string,

logoffUrl: string;

// this can be a full url eg http://www.google.com

postLogoffUrl: string;

defaultPageSize: number; // can be overridden by server

listCacheSize: number;

shortCutMarker: string;

urlShortCuts: string[];

keySeparator: string;

objectColor: string;

linkColor: string;

autoLoadDirty: boolean;

showDirtyFlag: boolean;

defaultLocale: string;

// caching constants: do not change unless you know what you're doing

httpCacheDepth: number;

transientCacheDepth: number;

recentCacheDepth: number;

// checks for inconsistencies in url

// deliberately off by default

doUrlValidation: boolean;

// flag for configurable home button behaviour

leftClickHomeAlwaysGoesToSinglePane: boolean;

logLevel: "error" | "warn" | "info" | "debug" | "none";

## Customising the generic client using Angular

If you wish to customise the user interface beyond what is possible through simple configuration (above), you will need to build the NOF client from source code. The best way to learn how to do this is create a new, separate, Client project, either in place of the existing one or alongside it. We will assume the latter approach in the explanation below.

As a pre-requisite you will need

* The Angular/CLI including its own pre-requisites (e.g. NPM). See <https://github.com/angular/angular-cli>

**Important**: Install the version of @angular/cli that is specified in the package.json file under devDependencies.

When this is all installed, from a command prompt *with Administrator permission* where you want your client project to be created:

* ng new MyProjectName

If prompted for ‘Angular routing’, answer N for no, and for the ‘stylesheet format’ select CSS.When finished (this could take several minutes depending on your network speed) and you are back at the command prompt,

* cd MyProjectName

Then install each of these packages in turn:

* npm install @nakedobjects/schematics –save-dev
* npm install @nakedobjects/gemini –save-prod
* npm install @nakedobjects/cicero –save-prod
* npm install @nakedobjects/view-models –save-prod
* npm install @nakedobjects/services –save-prod
* npm install @nakedobjects/restful-objects –save-prod
* npm install @angular/cdk –save-prod

Then

* ng generate @nakedobjects/schematics:new-project

When finished, edit the appPath property config.json file to point the appPath property to the Url for the server.

Then, from the command line again, specifying the port to run on e.g.

* ng serve --port 5001

**Note**: Whatever port number you choose (e.g. 5001 in the example above) you will need to configure CORS on the server to allow requests from that source.

(To stop the server, type **Ctrl-c** in the Command Prompt – twice if necessary.)

To toggle the CSS between the standard and ‘alt’ files, use this command:

* node node\_modules/nakedobjects.spa/togglecss.js

Run it again to toggle back to the generic CSS.

**Important note:** For those familiar with Angular/CLI, it should be noted that Naked Objects does not currently support ‘ahead of time’ (AOT) build.

### Adding custom views

To add custom views you will need to build the client using Angular/CLI and Node Package Manager as described [above](#_Adding_authorization).

Writing custom views requires some understanding of TypeScript and of the Angular framework. However, the following script will help you get started. Let us say that you require a simple customisation of the view for the Product object.

1. From the *elevated* (administrator) command window on your client project

* ng g c Product

This will create a Product directory, which you can browse via Visual Studio Code. The new folder should contain skeleton versions of the following files:

product.component.ts (TypeScript file containing the ProductComponent class)

product.component.spec.ts

product.component.html

product.component.css

1. Again, navigating within Visual Studio Code, locate the file app.module.ts, and add ProductComponent into the list of ‘entry components’, thus:

entryComponents: [

ObjectComponent,

ListComponent,

ErrorComponent,

ProductComponent

],

1. Now implement the ProductComponent class (in the product.component.ts file). For a trivial customisation you can simply inherit from the generic ObjectComponent class. You will need to copy the constructor from ObjectComponent, in order to ensure that all necessary services are injected, add import statements as needed, and *ensure that you are not accidentially overriding the* ngOnInit function. For example:

import { Component, OnInit } from '@angular/core';

import {ObjectComponent, UrlManagerService, ContextService, ViewModelFactoryService,   
 ColorService, ErrorService,ConfigService} from 'nakedobjects.spa';

import {ActivatedRoute } from '@angular/router';

import { FormBuilder} from '@angular/forms';

@Component({

selector: 'app-product-component',

templateUrl: './product-component.component.html',

styleUrls: ['./product-component.component.css']

})

export class ProductComponentComponent extends ObjectComponent {

constructor(activatedRoute: ActivatedRoute, urlManager: UrlManagerService,   
 context: ContextService, viewModelFactory: ViewModelFactoryService,  
 colorService: ColorService, error: ErrorService, formBuilder: FormBuilder,   
 configService: ConfigService) {

super(activatedRoute,urlManager,context,viewModelFactory,colorService,error,  
 formBuilder,configService);

}

~~ngOnInit() {~~ //delete this

~~}~~

}

1. Similarly, for a trivial example of customisation, you can simply copy the complete html from object.component.html, which you can find in this location:

* [projectDir]\node\_modules\nakedobjects.spa\lib\app\object

Into the new product.component.html, and then tweak the html, for example to give it a custom title.

1. Similarly, you can clone the styling from object.component.css into product.component.css.
2. Back in the elevated command prompt, enter:

* ng g s MyCustomConfig

which will create MyCustomConfigService.

1. Implement MyCustomConfigService (in the my-custom-config.service.ts file) by extending the existing CustomComponentConfigService, and specifically implementing the configureCustomObjects method, to relate the new component to the correct type in your domain model, for example:

import { Injectable } from '@angular/core';

import {CustomComponentConfigService,ICustomComponentConfigurator} from 'nakedobjects.spa';

import {ProductComponent} from './product/product.component';

@Injectable()

export class MyCustomConfigService extends CustomComponentConfigService {

configureCustomObjects(custom: ICustomComponentConfigurator): void {

custom.addType("MyNamespace.Product", ProductComponent);

}

}

1. In app.modules.ts, update CustomComponentConfigService provider

providers: [

UrlManagerService,

ClickHandlerService,

ContextService,

RepLoaderService,

ViewModelFactoryService,

ColorService,

ErrorService,

MaskService,

CustomComponentService,

// to configure custom components create implementation of   
 ICustomComponentConfigService and bind in here

{ provide: CustomComponentConfigService, useClass: MyCustomConfigService },

1. Run the application via e.g.:

* ng serve –port 5002

and test that your trivially customised view is used for the Product object type.