|  |  |
| --- | --- |
|  | **Unidev** |
|  | **Unified Development, Inc.**  16690 Swingley Ridge Road  Suite 260  Chesterfield, MO 63017  (636) 532-4424 |

|  |
| --- |
|  |
|  |

**Duncan Solutions, Inc**.

Kendo Grids

November 2013

Revision 1.00

**Prepared by:**

Ron Howard

Caleb Miller

Unified Development

**Table of Contents**

[1. Overview 4](#_Toc373906866)

[1.1 Goal 4](#_Toc373906867)

[2. Target Audience 5](#_Toc373906868)

[3. Grid Types 6](#_Toc373906869)

[3.1 Determining Grid Type 6](#_Toc373906870)

[4. Functional Requirements 7](#_Toc373906871)

[5. Locked Columns 8](#_Toc373906872)

[6. Hiding / Reordering / Renaming Columns 10](#_Toc373906873)

[6.1 Customer Grids 10](#_Toc373906874)

[6.2 Using the Data 11](#_Toc373906875)

[Data Access 12](#_Toc373906876)

[6.2.1 Process 12](#_Toc373906877)

[7. Exporting 15](#_Toc373906878)

**Revision** History

|  |  |  |  |
| --- | --- | --- | --- |
| Revision | Author | Description | Revision Date |
| 1.00 | Caleb Miller | Initial version | 11/26/2013 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# Overview

## Goal

The purpose of this document is to explain the concepts and methodologies used when with Kendo within the system. Duncan needed a specific set of functionality for the PEMS project that wasn’t out of the box for Kendo. This document will explain what those rules are and how Kendo had to be modified to meet the functional requirements.

The following items will be covered:

1. Grid Types
2. Functional requirements
3. Locked Columns
4. Hidden / Reordering / Renaming Columns
5. Data Access - Performance
6. Exporting

# Target Audience

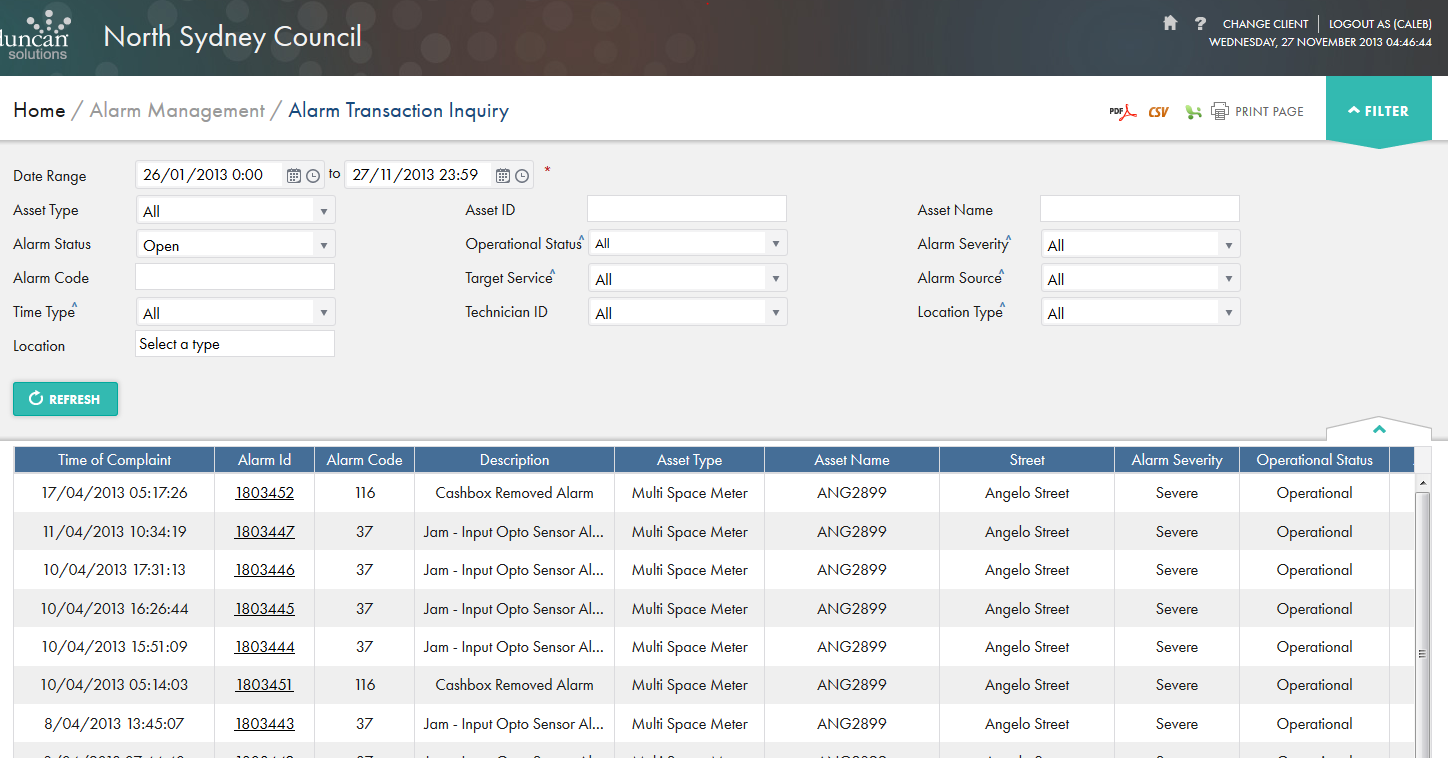
The target audience of this document is person or persons who have:

1. Experience in the following technologies:
   1. Javascript / Jquery. This is used heavily on each grid.
   2. C# /.Net / MVC
   3. Working knowledge of Telerik and Kendo MVC
   4. T-SQL, SSMS, SqlServer, Stored Procedures and Views
2. The user of this document has a full understanding of the Duncan PEMS project. This includes database and system architecture knowledge, desired business rules of the application, etc.
3. Microsoft SQL Server administration and understand rights, database creation and administration, and are able to use either SSMS or SQL command line interface. The user will need the ability to create and maintain stored procedures and views.
4. Visual Studio 2012

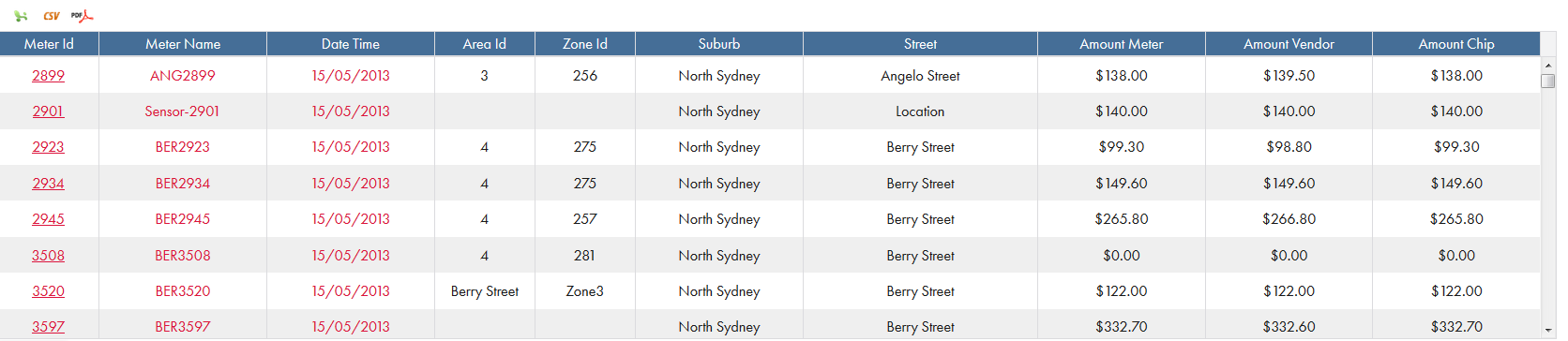
# Grid Types

There are two basic types of grids we use in the system: Index page grids (Complex) and Details page grids (Basic). Complex grids typically use all or most of the required functionality and basic grids are simpler that use a small subset of the possible functionality. This document will explain the Complex grids and will be using the Alarms Grid as an example, since it uses all of the possible functionality and is one of the more complicated grids in the system.

Below is an example of the Complex grid for Alarms.



Here is an example of a Basic grid (from the Collection Aggregation Details page)



## Determining Grid Type

While you can use the Complex grid for every grid in the system, if a grid is simple and doesn’t require complex functionality (See Functional Requirements Below), then use a basic grid. This will drastically increase development time and maintainability in the future. Typically a basic grid will take about 1/3rd to ½ the time to develop and is easier to test. If the grid requires any of the following functionality then it is considered a complex grid: Heavy performance considerations, Locked Columns, Reorder Columns, and Hidden Columns. The Users and Roles index pages are examples of a basic grid type, and the Alarms, Events, Transactions index pages are all examples of Complex grid types.

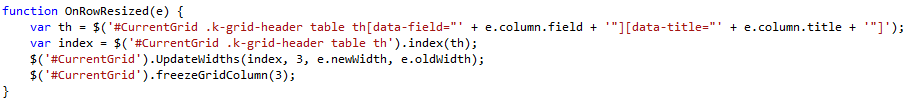
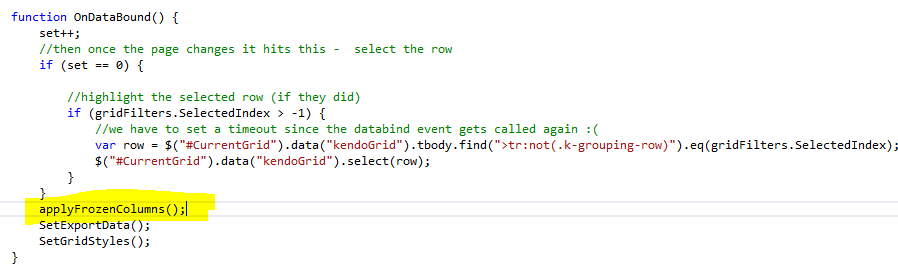
# Functional Requirements

Below is a list of requirements that each grid has to be able to support. Not all grids will use all of the listed functionality.

1. Hidden Columns – individual columns will need to be hidden. This is per grid / per customer
2. Re-Order Columns- need the ability to re-order columns on each grid. This needs to be customer specific as well.
3. Locked Columns – X number (2, 3, 5, etc.) of left most columns must be locked. This is grid specific, but not customer specific. For example, the alarms grid has two locked columns, so the two left most columns must remain where they are when the user is scrolling.
4. Localization – all of the grid data (phone numbers, dates, times, and currency) must be localized to the current customer.
5. The grids for index pages should not be displayed until the user hits the “Refresh” button. This is to get them to use the filters to narrow down the search before they see results.
6. Export functionality. Each grid needs to be able to be exported in Excel, PDF, and CSV. The exports must respect all of the filters provided, and for the PDF and Excel, the chosen filters must be displayed at the top of the file. Also, the custom titles, orders, etc should be displayed correctly.
7. Paging, sorting, standard grid functionality: We will be out of the box functionality that the Telerik Kendo MVC grids provide. Page sizes have been set to 100, 250, 500, and 1000 for performance considerations.
8. Filters should be saved when navigating to a details page and returning to an index page. This is done by storing the current filters in session upon navigation to the details page, and upon return to the index page, checking to see if this filter data exists. If it does, pre-select all the filters and load the grid.

# Locked Columns

In order to lock a column in the system you will need to apply methods from the locked column gridExtension.js file located in the /Scripts folder of the web root. This file is an extension of the grid. It relies on knowing the width of each column and the current position of the users horizontal scroll bar to determine what columns to show / hide. All the tool is doing is hiding or display columns conditionally based on the total width of the grid and the current position of the horizontal scrollbar. Here are the steps to lock columns for a grid (all of the examples shown are on the /Areas/city/Views/Alarms/Index.cshtml file):

1. There are two methods you will use: UpdateWidths and freezeGridColumn.
   1. freezeGridColumn will determine the widths of all the columns and perform the required logic to lock the amount of columns passed in. This is required to lock the grid to the current total grid width. This will need to be called on grid events (databind / rowResized) and when the user resizes the browser window, as this effects the total width of the grid.
   2. UpdateWidths will update the existing widths of the columns and re-lock the grid with those new widths. This is required when a user changes the width of a column.
2. You must re-size and re-lock the columns when the browser window size changes. Typically this is done in the documents “ready” event.
   1. 
3. When you resize a column width, we need to update the grid extension variables and re-lock the columns based on the new widths. This is done by calling a few extension methods when the Kendo event for Column Resizing and Column Data-bound is fired off.
   1. 
   2. 
   3. 

Couple of other items to keep in mind:

1. Locking columns is grid specific and NOT customer specific.
2. Every column must have a width assigned.
3. The columns lock the left most N number of columns, so if we are locking 3 columns then the left most 3 columns will be locked.
4. Not all grids have locked columns (User Index, Role Index)

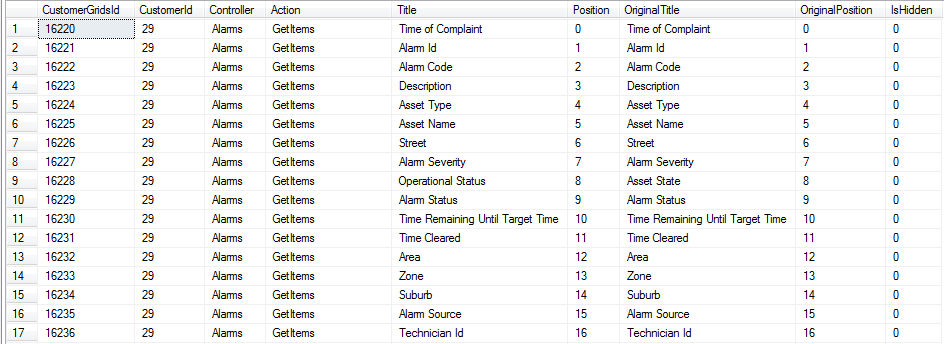
# Hiding / Reordering / Renaming Columns

In order to conform to the requirement of hiding / reordering, and renaming columns, we devised a process in which customers can define the look and feel of the grid and the grid will respect those settings. This section only applies to the Complex grids and the Basic grids will not have any associated data described in this section. In order to export Basic Grids, see the Users or Roles Index pages “Exporting” region and follow those examples, as they are straightforward.

## Customer Grids

Inside the RBAC DB there is a table called Customer Grids. This table holds all of the customer specific Complex grid column data and allows us to determine the order in which a specific customer wants to display each grid, the titles of those grids, and which columns should be displayed to the user (or hidden). There is a certain process that is involved in setting up this data and maintaining it, but for the purposes of the grids and this document, they just use this specific table.

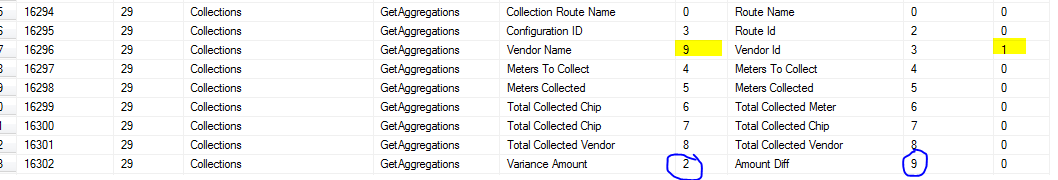
Here is the North Sydney Council -> Alarms Index page data:



The specific grid data is identified by the unique Customer Id, Controller, and Action. Each grid will contain a list of every column on the grid, the desired position, the position that the grid used by default (original position), any custom column header title needed, and if the column is hidden or not.

Items of note:

1. Hidden columns must be the last positioned elements. So if you want to hide 3 columns in this example, 16, 15, and 14 would be denoted as hidden.
2. The original position has to correspond to the position it was placed on the grid within the View.
3. There cannot be duplicate position numbers, each one must be unique.
4. Here is an example of the collection aggregation for a customer that is using a hidden column and a different ordering:

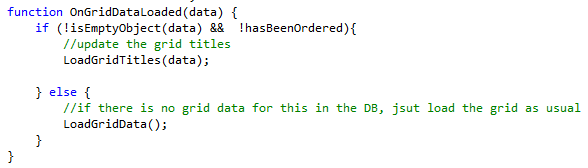


## Using the Data

The base controller has a method called GetGridData that will return a list of columns with the correct titles in the correct order for a customer, controller, and action. We call this method to get a list of all the columns for this grid in order and with the correct titles. We do this from the client side:



Once we get a return from that method, we check to see if we got any data back.



If we didn’t, then this grid does not have custom columns, so we load the grid data using whatever it defaults it to. If something does come back, then we need to modify the grid bound to the page to use the data that came back from the DB.

1. First we load the titles for the grid by replacing the current column headers with the Titles for the columns we got back.
   1. 
2. Then we re order all of the columns to conform to the positions defined in the result set we got back from the DB
   1. 
   2. This does some comparative logic using the original position and new position to move columns around and place them in the correct position.
   3. It also hides the last N number of columns using a gridExtensions.js method.
3. And finally we load the grid data (see Data Access section below.)

# Data Access

There are two ways we access data for the grids in the system: Method and Query syntax via Entity Framework (the ORM used in the system) and by using Stored Procedures and Views. The users and roles index pages have examples of the method / query syntax, as well as collection aggregation index grids. Those display standard ORM data access via Entity Framework, so this document will not cover those items.

The stored procedure version is a bit more complicated and is used for the grids that are data intensive and performance is a concern (querying tables with millions of rows, where indexes are required, etc.). Basically the stored procedure version builds a list of SQL parameters from the filters on the grid, passes them to a stored procedure which in turn builds a dynamic “Where” clause that selects against a view that brings back the data that represents the grid data. So there are 3 main components that you have to understand: Filters, stored procedure, and the view. We are using the Alarms Index grid as an example.

### Process

1. First, we have to pass all of the correct filters to the action method. We do this by building the filters on the page via jQuery, assigning these filters to the data request for kendo, and called the grids data source Read method. This will take the filters provided (along with page size, page number, etc) and build a data source request object and pass it off to the Action method. You can see an example of the filter building in the LoadGridData method on the Alarms Index page



1. Now that we have the filters in the action method, we have to build a list of SQL parameters that the stored procedure will accept. All of the filters reside within the request object provided in the action method signature:

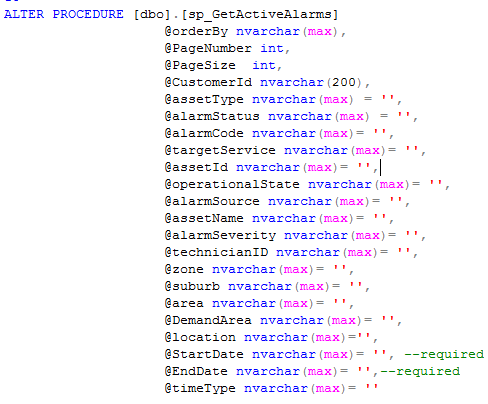
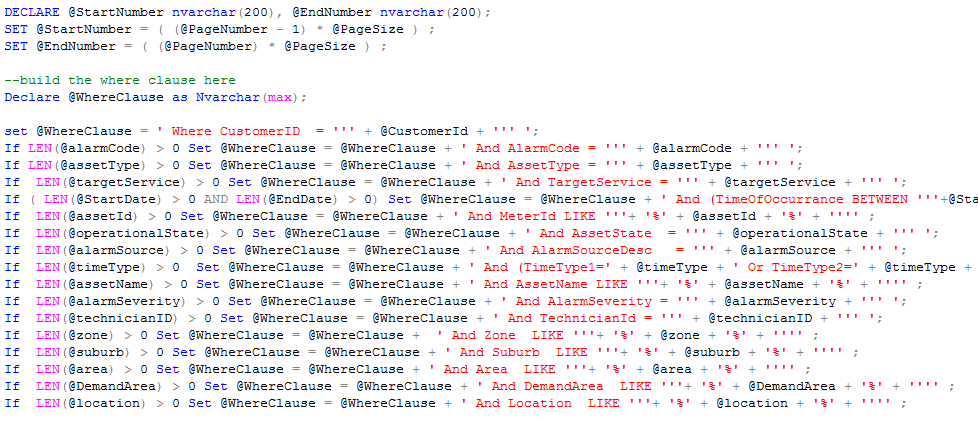


1. We build the filters by calling the “GetSpParams” method of the base factory with a default order by for that grid. The default is overridden if the user has sorted the grid and this data is already contained in the data source request.



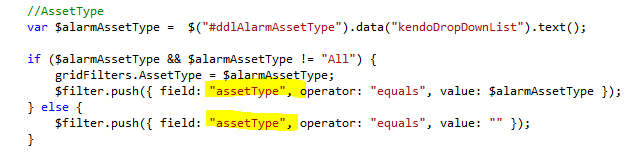
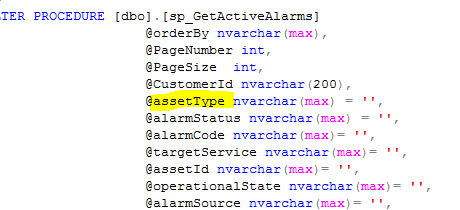
1. This will take the filters from the data source request and build a list of sql parameters that the stored proc will use. Since these filters come from the client side, all of the SP parameters need to be passed in on the client side IN ORDER for this to work. There are also a few sql parameters that are not passed in client side (page size, page number) that are added based on the data source request object.
   1. If you have a parameter in the stored proc, but no filter value selected by the user, pass an empty value so the parameter gets generated and the stored proc works as intended.
2. Once we have our filters we pass them to the stored proc to get the data.



1. Here are the parameters for the Alarms stored proc:
   1. 
   2. The parameters coming from the client side will need to be IN THIS ORDER and have the required fields in order for it to work. Most have start and end times as required for performance reasons.
2. Now we generate the dynamic where clause we will run against the view: 
3. As you can see, we are only adding to the where clause when the parameter passed in had a value. This allows the application to be able to filter on any number of values passed in and still work correctly.
4. Then we select against the view to bring back all the appropriate data. The view is a representation of the grid data without any filters applied to it, so all the stored proc is doing is filtering the grid data at the DB level so we don’t have to do it in memory.

Items of note:

1. There are two types of where clauses we are creating, likes and =’s.
2. orderBy, PageNumber, PageSize, and CustomerId are always required
3. StartDate and EndDate are required for performance reasons in most grids
4. If the grid is not performance heavy it is recommended that you use the Method / Query syntax provided by Entity framework, as it will cut down on development time and cost.
5. Review pv\_ActiveAlarms view in the DB for more specifics on view creation.
6. All views need to return RowNumber so the stored proc can filter correctly.
7. All rows will need to return a count so we can tell the grid the total filtered count.
8. The names of the filters built on the client side have to match the Name of the stored procedure parameter. So for alarms, if the property you want to filter on is Asset Type, then the filter we build in the LoadGridData method has to match the stored proc parameter: alarmType

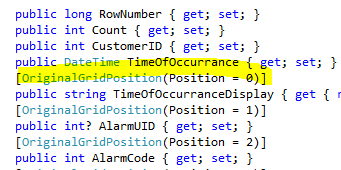


# Exporting

There are two types of grid exports, one for basic and one for Complex. Basic examples can be seen in the Users and Roles Index grid exports under the “Exporting” region.

Complex examples take a bit more work, since the export has to respect the custom titles, orders, hidden columns, etc that the grid did. In order to achieve this we have a process that will take the objects that the grid represents (the data bound to the grid) and dynamically generate a csv, pdf, or excel file that has the properties of the object ordered, titled, and hidden according to the Customer Grids table.

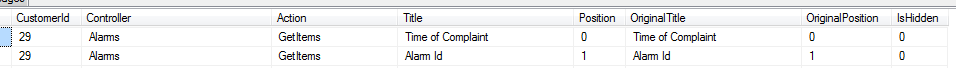
The object bound to the grid is a representation of the grid data. For alarms, it is called SpAlarmModel and is found in the AlarmModels.cs class in the Entities project. Here is a snippet of the alarm object we are discussing:



Notice the Original Grid Position attribute assigned to the different properties of the object. For each column in the grid, the associated property MUST have this attribute on the corresponding property used in the grid column.



As you can see, Time of Occurrence is the first column and it has the original position attribute set to 0. Alarm UID is the second column, and its original position attribute is set to 1. This will also correspond to the data in the Custom Grids table:



This concept is how we order the export dynamically, since we basically have to re-order, re-title, and hide the columns just like we did on the client side, but this time on the server. An export factory has been created to do just that. The PDF and Excel methods add a few filters to the top of the file, but otherwise functions the same as the csv export.

Now the same process applies as Data Access. We build the filters, call the stored proc, and return the data we need. Now that we have a list of objects and a data source request, we will use the Export Factory to call the appropriate export generation method. We will pass in the list of objects, the controller, action, and the customer Id.

This method takes the list of objects passed in (Alarms in this example) and determines all of the properties the grid cares about (based on the Original Grid Position attribute).

Then it gets all of the grid data from the DB for this controller / action / customer. Some comparison logic is used to determine all of the grid data in the correct order to use for the export generation.

The result of this is an ordered, titled, valid set of grid data that represents the grid the user saw on the screen in the same order with the same columns displayed. Once we have this list, we can generate the appropriate export file and return the memory stream back to the user for download.

This is the code that generates a list of properties for an object sorted correctly so we know how to correctly generate the export file column order:

