Contents

[Configuring your Application to Run Foundation 3](#_Toc381184062)

[Overview 3](#_Toc381184063)

[Members of The IFoundationConfigurator interface 3](#_Toc381184064)

[Configuring Web Features 3](#_Toc381184065)

[Form Builder 4](#_Toc381184066)

[Overview 4](#_Toc381184067)

[ViewModel Attributes 4](#_Toc381184068)

[UI (View) Level 5](#_Toc381184069)

[Paging: 6](#_Toc381184070)

[Overview: 6](#_Toc381184071)

[General Notes: 6](#_Toc381184072)

[STEP1: Preparing your view model for paging: 6](#_Toc381184073)

[STEP2: Preparing your Model Populator (Query Layer): 6](#_Toc381184074)

[STEP3: Preparing your controller action method 6](#_Toc381184075)

[STEP4: Render Paging Links 7](#_Toc381184076)

[Sorting: 8](#_Toc381184077)

[Overview: 8](#_Toc381184078)

[Step1: Preparing your View Model for Sorting 8](#_Toc381184079)

[STEP2: Preparing your Model Populator (Query Layer): 8](#_Toc381184080)

[STEP3: Preparing your controller action method 8](#_Toc381184081)

[STEP4: Render Sortable Headers 9](#_Toc381184082)

[Results Filtering 10](#_Toc381184083)

[Overview 10](#_Toc381184084)

[Creating the Filter Box 10](#_Toc381184085)

[Applying Filters to a Query 10](#_Toc381184086)

[AJAX Features – Show Link Contents in a Floating Popup 12](#_Toc381184087)

[STEP1: Preparing your Layout for AJAX Features – Include JS File 12](#_Toc381184088)

[STEP2: Loading Screen 12](#_Toc381184089)

[STEP3: Changes to Controller Action 12](#_Toc381184090)

[Displaying Link Contents in Modal Popup 12](#_Toc381184091)

[Showing a link Contents in a popover div 13](#_Toc381184092)

[Configuring the Floating Popup 14](#_Toc381184093)

# Configuring your Application to Run Foundation

## Overview

Foundation features are configured through the properties of FoundationConfigurator you can create instance of this class or you can provide your own implementation that implements the interface IfoundationConfigurator. Pass your configurator object as a paramaeter to the static method : FoundationKickStart.Configure

public static void Configure(Foundation.Configuration.IFoundationConfigurator *foundationConfigurator*)

Member of Foundation.Configuration.FoundationKickStart

You should place the call to FoundationKickStart.Configure(IfoundationConfigurator) in your global.asax.cs for your web application or in the main routine of your console application.

## Members of The IFoundationConfigurator interface

The properties of this interface are described in the table below

|  |  |  |
| --- | --- | --- |
| Member | DataType | Description |
| UserBusinessManager | Bool | Whether the Business Managers Container Feature is used in the project or not. For more information, refer to the section detailing the Business Managers Features of Foundation |
| UseEmailing | Bool | Whether your application uses the email sending features of Foundation. For more information, refer to the section detailing the Emailing Features of Foundation. |
| UseSecurity | Bool | Whether your application is using the Security and Authentication features. |
| UsePersistence | Bool | Whether the persistence features are used. |
| UseQueryContainer | Bool | Whether the Query Container features are used. |
| Web | WebConfigurations | Configuration Parameters for web features |
| Persistence | PersistenceConfigurations | Configuration Parameters for Persistence features |
| Business | BusinessConfigurations | Configuration Parameters for business Features |
| Mongo | MongoConfigurations | Configuration Parameters for Mongo DB support features |

## Configuring Web Features

Web Features are configured through the class WebConfigurator

* **public Type ControllersAssemblyHookType**Pass a type of one of your MVC Controllers (using typeof(MySampleControlelr) ) as pointer to the assembly hosting your MVC controllers. Foundation will scan the assembly that contains this type and map all the classes that implements the interface IController and associate them with the custom controller factory. Using the custom controller factory rather than the default one that comes with the ASP.NET MVC allows injecting the controller dependencies through Structure Map when instantiating your controller object (Like injecting business manager containers and Query container for example).
* **public Type AuthenticationService { get; set; }**A pointer to the Authentication service which handles security and authentication.
* **public Type FlashMessenger { get; set; }**The flash messenger to be used in communicating messages between the business layer and User interface. A type that implements (Foundation.Infrastructure.Notifications.IFlashMessenger)
* **public ResourceManager FlashMessagesResourceManager { get; set; }**the resource manage for the flash messages. If you store you Flash Messages in a resource File called “ErrorMessages” , then you need to assign this property to ErrorMessages.ResourceManager
* **public ResourceManager FlashMessagesResourceManager { get; set; }**the resource manage for the flash messages. If you store you Flash Messages in a resource File called “ErrorMessages” , then you need to assign this property to ErrorMessages.ResourceManager
* **public ResourceManager PageTitleResourceManager { get; set; }**Resource manager for page titles. the key in the resource file should be the name of the view model omitting the suffix "ViewModel" if it exists.
* **public string DefaultPageTitle { get; set; }**  
  the default page title if nothing specific is provided in PageTitleResourceManager.
* **public ResourceManager HelpResourceManager { get; set; }**

Resource manager for help content. the key in the resource file should follow the following format ViewModelTypeName\_ElementName.

# Form Builder

## Overview

This component provides the functionality for form auto-generating according to the View Model declaration. By specifying attributes for the properties of the view model, you can tell the Form Builder what control type to render.

This component uses Twitter Bootstrap html layout and classes.

## ViewModel Attributes

The main attribute that indicates that the property needs to be rendered as a editable control is EditControl. It has following parameters:

* ElementType – an enum specifying the control type to be rendered. Potential values are:
  + Text (for a simple text box)
  + Hidden (for a hidden field)
  + TextArea (for a text area)
  + Password (for a password field, like for a registration form)
  + WholeNumber (in read-only mode, the data is formatted as a whole number)
  + FloatingPointNumber (in read-only mode, the data is formatted as a floating-point number)
  + DateTime (in read-only mode, the data is formatted as a DateTime, in edit mode it is rendered as a text box with class „datepicker”- you can write your custom javascript to add calendar controls to all inputs with class „datepicker”)
  + Time (in read-only mode, the data is formatted as a Time)
  + CheckBox (for a check box)
  + Enum (for a drop down with an enum as a selection type)
  + List
  + ListBox
  + Guid
* Cols – if the ElementType is TextArea, then this property is translated into „cols” attribute of text area (default = 60)
* Rows – if the ElementType is TextArea, then this property is translated into „rows” attribute of text area (default = 6)
* MaxLength – if the ElementType is Text, then this property indicates the maximal length of the text box.

Foundation FormBuilder also supports native .NET attributes like [Required] and [Display(Name = "Name")]. The Required attribute will instruct the FormBuilder to add client-side validation attributes (if the jQuery Unobtrusive validation is enabled by adding a script reference on the UI level). The Display attribute with its parameters allow you to specify some appearance parameters, like order in which the control will appear, group name, the control label, etc..

So, to define a control that will appear as a TextBox, your view model property could look like this:

[Display(GroupName = "Contact Info", Order = 2, Name = "Email")]

[EditControl(ElementType = ElementType.Text)]

[Required]

public string Email { get; set; }

## UI (View) Level

The point of this component is to simplify the creation of forms so that for the most part you won’t need to explicitly implement your forms, but instead use the following structure:

@using (var form = Html.Bootstrap().BeginForm(BootstrapFormType.Horizontal, new {@class="well", id="autoForm" }))

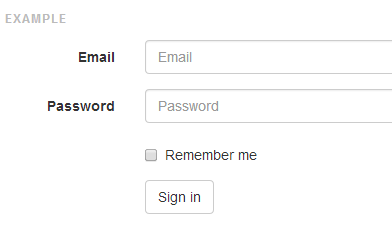
{

@Html.DynamicForm()

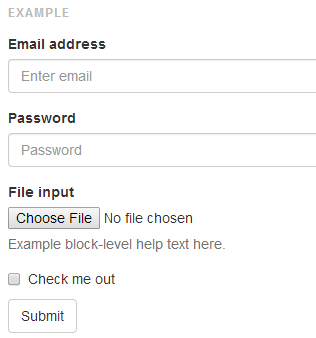
}

This will take your view model, analyze its properties and their attributes and accordingly generate a form and edit controls. By specifying BootstrapFormType Enum value, you instruct the Foundation to apply specific styling rules (class) to the generated form. There are three possible values:

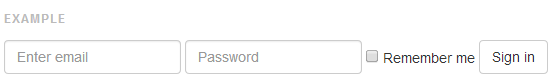
1. **BootstrapFormType.Horizontal:**



1. **BootstrapFormType.Vertical:** (this is the default style)



1. **BootstrapFormType.Inline:**



# Paging:

## Overview:

Implementing paging involves several steps:

* When the user jumps to a certain page, the code needs to pass the relevant information to the controller while maintaining any other query string parameters that are used to filter the results and sorting them.
* Select only a subset of rows relevant to the page the user wants to display.
* Enrich the View Model with paging information so that the View can be rendered correctly (current page, total number of results, total number of pages ...).
* Render the paging links in the view.

The Foundation project helps you across all these steps.

## General Notes:

* Default Page Size: “Foundation\_PageSize” configuration key in AppSettings. If absent, the default value is 10 records per page.

## STEP1: Preparing your view model for paging:

* **The view model should have a member implementing the interface *Foundation.Web.Paging.IPagingParameters.***
* **You need to ensure that the view model constructor populates this member with a default instance.**
* A Quick way to get a model of this type is to make your view model inherit from Foundation.Web.Paging.PagedViewModel. This will add a member PagingInformationViewModel to your view model.

## STEP2: Preparing your Model Populator (Query Layer):

* An extension method FetchPaged is added to any IQueryable.

public static IPagedList<T> FetchPaged<T>(this IQueryable<T> query, IPagingParameters pagingParameters)

* This method expects only one external parameter which is IPagingParameters. It should be a member of your model already (refer to step 1).
* The return type of this method IPagedList<T> represents an IList<T> which contains the items to be displayed in the current page plus another member PagingViewModel which contains the information required for rendering the paging links.
* You then need to copy the new paging information to your view model before sending it to the view engine. To do so you can make use of the following extension method

public static IPagingParameters FillPagingParameters(this IPagingParameters destination, IPagingResults parameters)

* This extension method is available for any member that implements IPagingParameters and it simply copies all the paging information in one line of code.

## STEP3: Preparing your controller action method

Make sure that one of your action parameters is of type IPagingParameters.

You need to decorate your action with the following Action filter [RenderPagedView], this will ensure that to objectives are achieved:

1. Your Model will be enriched with controller/action information necessary to render links
2. Your Model will be populated with default paging parameters if none are sent in the query string.

## STEP4: Render Paging Links

Use the HTMLHelper extension method to render the paging links.

public static MvcHtmlString PageLinks(this HtmlHelper html, object queryObject, int linksToShow = 0)

# Sorting:

## ­Overview:

Similar to Paging, adding sorting capabilities consists of several steps:

* The User’s Sorting Intention (Sort Column and Direction) is captured from the view and passed to the action.
* The Query Layer Applies the sort to the list before applying any paging.
* The Model carries information about the sorting parameters used back to the view to render the relevant column headers.
* Sorting needs to be maintained if the user jumps from a page to another.

## Step1: Preparing your View Model for Sorting

* **The model should have a member implementing the interface *Foundation.Web.Paging.ISortingParameters.***
* **You need to ensure that the view model constructor populates this member with a default instance.**
* A Quick way to get a model of this type is to make your view model inherit from Foundation.Web.Paging.PagedViewModel. This will add a member PagingAndSortingParameters to your view model (it contains the sorting properties along with paging ones).

## STEP2: Preparing your Model Populator (Query Layer):

* An extension method ApplyOrder is added to any IQueryable by referring to the namespace Foundation.Web.Sorter

public static IQueryable<T> ApplyOrder<T>(this IQueryable<T> source, ISortingParameters sortingInfo)

* This method expects only one external parameter which is ISortingParameters. It should be a member of your model already (refer to step 1).
* The return type of this method IQueryable<T> represents your results sorted according to the right key and sort order.
* You then need to copy the new sorting information to your view model before sending it to the view engine. To do so you can make use of the following extension method

public static ISortingParameters FillSortingParameters(this ISortingParameters destination, ISortingParameters parameters)

* This extension method is available for any member that implements ISortingParameters and it simply copies all the sorting information in one line of code.

## STEP3: Preparing your controller action method

Make sure that one of your action parameters is of type ISortingParameters.

You need to decorate you action with the following Action filter [RenderPagedView], this will ensure that to objectives are achieved:

1. Your Model will be enriched with controller/action information necessary to render links
2. Your Model will be populated with default sorting parameters if none are sent in the query string.

## STEP4: Render Sortable Headers

Use the HTML Helper extension method to render the column header.

public static MvcHtmlString SortableHeader(this HtmlHelper row, ISortingParameters sortingInfo, string columnId, string title, object htmlAttributes = null)

When the user clicks this header the following events takes place:

* GET request is sent to the relevant controller-action with the data element to sort with *SortColumn* & the new direction as *SortDirection*.
* Columns that can be used in sorting but not used at the moment will be given the CSS class specified in *SortableHeaderCssClass*
* The clicked column is highlighted as the current sort using the css class : SortedHeaderCssClass
* A Glyphicon corresponding to the current sort is displayed. Glyphicon Names are specified in the properties: *SortedIcondAscending* and *SortedIcondDescending.*

Note: The variables mentioned above are members of Foundation.Configuration.WebConfiguration.PagingConfigurationsproperties. For more information please refer to the section: Configuring and Bootstrapping.

# Results Filtering

## Overview

Result filtering process involves the following actions



## Creating the Filter Box

To create a filter box, you create a view model as normal. Controls on this view model will be mapped to properties in your Data Model. To instruct *Foundation* about how to do this mapping you need to decorate your Model’s properties with *FilterControl* attributes. This custom attribute is part of the name space *Foundation.Web.CustomAttribute*;

For example, below is a view model that maps a view model field “EmailAddress” to the Field *ContactEmail* of the Entity *Customer* which is a child of the Entity *Order* In your Query Type. As you can notice, you can reach children of children by just separating them by dots (.) as you would do in a LINQ query.

public class UserFilterModel

{

[FilterControl (DataElement = "Order.Customer.ContactEmail", CaseSensitive = false, OperatorOption = Operator.Like)]  
public string EmailAddress { get; set; }

}

The comparison operator to be used in the example above is the “Like” Operator and the value of this field will be used in a case-insensitive comparison. Operators that can be used are : Equal, Unequal, LessThan, LessThanOrEqualTo, GreaterThan, GreaterThanOrEqualTo and Like.

## Applying Filters to a Query

After the user have filled in the filter form and it has been sent back to your controller-action,

1. Make sure you have an instance of the ViewModel (the one you have designed in the step above) populated with the filter values.

On your IQueryable instance, call the Extension Method *ApplyFilter*. Which is available in the namespace Foundation.Web.Filter. This extension method expects only one Parameter which is an instance of the View Model.

* All the filter properties are AND-ed together.
* The Result of ApplyFilter on IQueryable is another IQueryable with a Where Clause applied.
* The Lambda expression that gets executed is Null-Safe.
* Null/Empty filters are ignored.

# AJAX Features – Show Link Contents in a Floating Popup

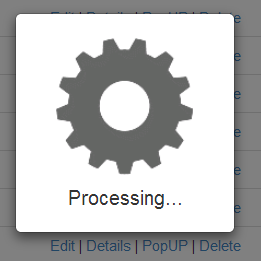
## STEP1: Preparing your Layout for AJAX Features – Include JS File

1. Include the Foundation.Shared.js file in your layout Page

## STEP2: Loading Screen

When executing lengthy Ajax Calls, it is recommended to show a loading screen to the user. The purpose of this screen is to give the user an indicator that his action has been captured and that the system is in the process of responding to it. The foundation Library shows the “in progress” window as a JQuery.modal before issuing the Ajax request and hides it when the request completes.

Your HTML Page needs to contain a DIV element with the ID=processing-modal. You can find an example of a processing modal here: <http://bootsnipp.com/snippets/featured/centered-processing-modal>



## STEP3: Changes to Controller Action

Make sure you action is returning a view through this method

return AdaptiveView("Edit", model);

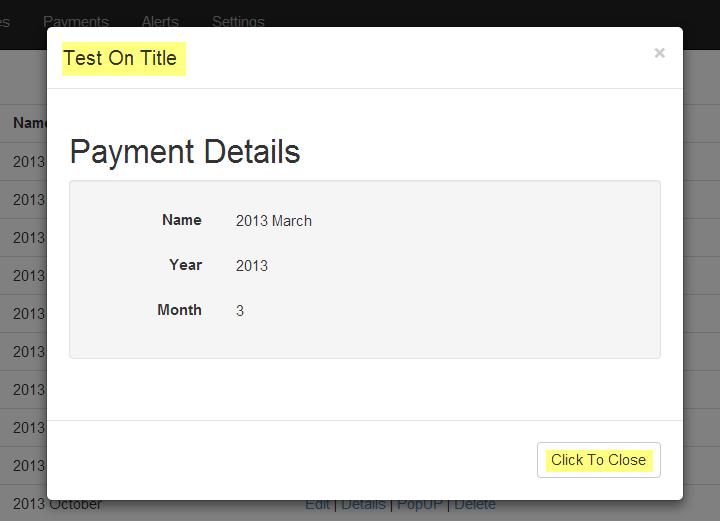
This method is available in the Foundation.Web.BaseController. This method returns a partial view for Ajax requests and returns a full view (with layout) for non-Ajax requests. This is useful if the user is browsing from a client that doesn’t support JavaScript or Ajax.

## Displaying Link Contents in Modal Popup

* Give the hyperlink this class “**showInPopUpOnClick**”.
* Add the following attributes to the link to customize the popup dialogue box

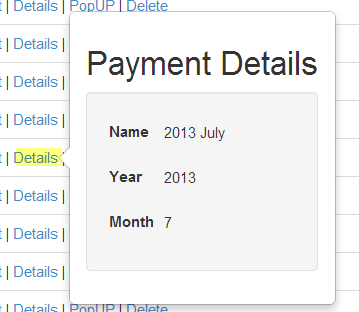
|  |  |
| --- | --- |
| Attribute | Description |
| data-popup-title | The title of the popup screen. Default is “Details”. |
| data-popup-dismissButtonText | The text for “Dismiss” Button. Default is “Dismiss”. |
| data-popup-hideButtonAndHeader | True (ture). Whether title bar and the dismiss buttons should be hidden. Default is false. |

Example:   
<a class="showInPopUpOnClick" data-popup-dismissbuttontext="Click To Close" data-popup-title="Test On Title" href="/PaymentPeriod/Details/f6603b8d-7eb7-4fd1-bd52-a29900cb7321">PopUP</a>



## Showing a link Contents in a popover div

* Give the hyperlink this class “showInBaloonOnMouseOver”.



## Configuring the Floating Popup

The HTML page must contain and HTML block (could be a DIV) with id=detailsModel. This HTML Block should contain elements with the following Id’s. A good place to put this block is in your \_Layout.cshtml page.

|  |  |
| --- | --- |
| ID | Description |
| detailsModelHeader | Container for the details popup header. |
| detailsModelTitle | Container for the Model Title.  The contents of this element will be filled with the value of data-popup-title. |
| detailsModelDimissButton | The buttons that dismiss button.  The text of this button will be set to the value of data-popup-dismissButtonText |
| detailsModelFooter | Container for the footer for popup. |
| detailsModelContent | This element will be populated with the response of the ajax call. |

Below is an example of a Pop Up.

<div class="modal" id="detailsModel">

<div class="modal-dialog">

<div class="modal-content">

<div class="modal-header" id="detailsModelHeader">

<button type="button" class="close" data-dismiss="modal" aria-hidden="true">&times;</button>

<h4 class="modal-title" id="detailsModelTitle">Modal title</h4>

</div>

<div class="modal-body" id="detailModelContent">

</div>

<div class="modal-footer" id="detailsModelFooter">

<button type="button" id="detailsModelDismissButton" class="btn btn-default" data-dismiss="modal">Close</button>

</div>

</div><!-- /.modal-content -->

</div><!-- /.modal-dialog -->

</div><!-- /.modal -->