Contents

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". If absent, the default value is 10 rows per page.

STEP1: Preparing your model for paging:

- **The 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 IQuerably.

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

- This method expects only one external parameter which is IPagingParamaeter. It should be a member of your model already (refer to step 1).
- The return type of this method IPageList<T> represents an IList<T> which is the items to be displayed in the current page plus another member PaginViewModel which contains the information required for rendering the paging links.
- You then need to copy the new paging information to your viewmodel 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 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. You 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, Applying sorting 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 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 Paging

- **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 `PagingInformationViewModel` to your view model.

STEP2: Preparing your model Populator (Query Layer):

- An extension method ApplyOrder is added to any IQuerably by referring to this 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 viewmodel 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. You 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:

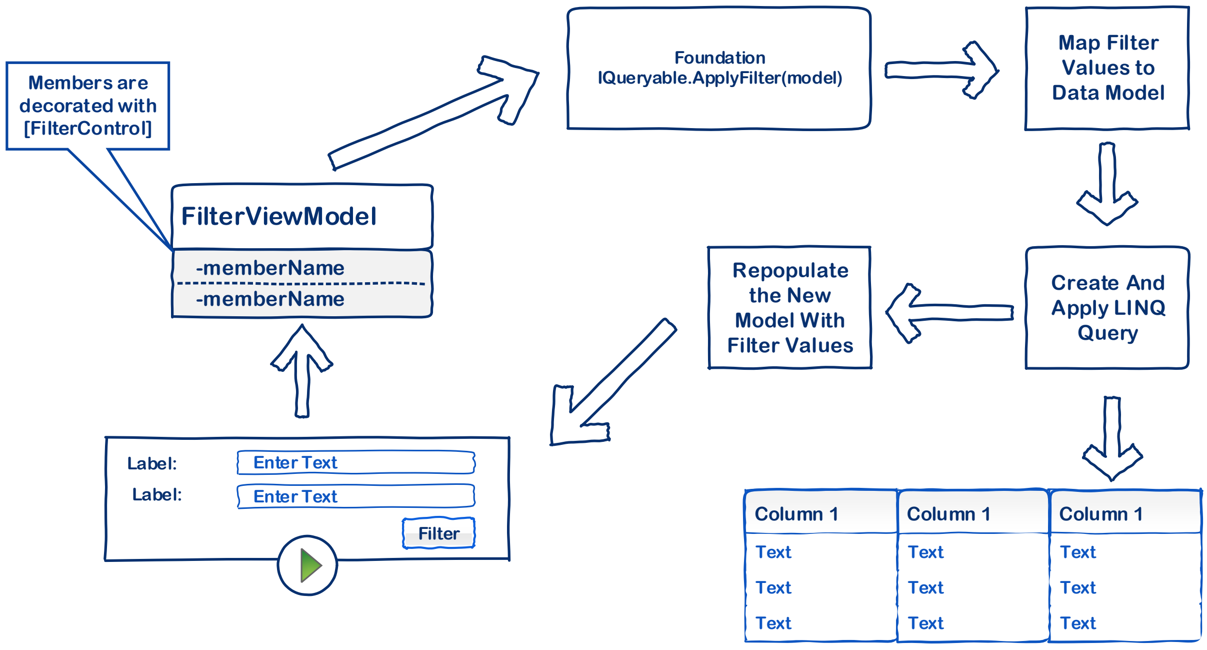
- GET request is sent to the relevant controller-action with the data element to sort with *SortColumn* & the new direction as *SortDirection*.
- Columns that can be used in sorting but not used at the moment will be given the CSS class specified in *`SortableHeaderCssClass`*
- The clicked column is highlighted as the current sort using the css class : `SortedHeaderCssClass`
- A Glypicon 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.`PagingConfigurations`*properties. For more information please refer to the section: Configuring and Bootstrapping.*

Results Filtering

Overview

Result filtering process involves the following actions


Members are decorated with [FilterControl]
FilterViewModel
-memberName
-memberName
Foundation IQueryable.ApplyFilter(model)
Map Filter Values to Data Model
Create And Apply LINQ Query
Repopulate the New Model With Filter Values
Label:
Enter Text
Label:
Enter Text
Filter
Column 1
Column 1
Column 1
Text

You have enough

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. Other 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 IQuerably is another IQuerable with a Where Clause applied.
- The Lambda expression that gets executed is Null-Safe.
- Null/Empty filters are ignored.