Pager Tutorial (v1.3 and earlier)

Note

This tutorial applies to Infusion v1.3 and earlier. If you are using Infusion v1.3.1 or later, please see the latest Pager Tutorial

This page will walk you through an example of adding the Infusion Pager to your application.

This tutorial assumes that:

- you are already familiar with HTML, Javascript and CSS
- you are familiar with what the Pager is and does
- now you just want to know how to add it to your file.

For more general information about the Pager, see Pager. For technical API documentation, see Pager API.

Tutorial: How to Use the Pager

Scenario

You're implementing a roster for a large school, and since there are thousands of students at this school, you want the list of students to be paged. This tutorial will show you how to use the Infusion Pager for this.

There are four basic steps to adding the Pager to your application:

- Setup: Download and install the Fluid Infusion library
- Step 1: Prepare your markup
- Step 2: Write the script, setting the options
- Step 3: Add the script and styles to your HTML

The rest of this tutorial will explain each of these steps in detail.

On This Page

See Also

- Pager
- Pager API

Still need help?

Join the infusion-users mailing list and ask your questions there.

Setup: Download and install the Fluid Infusion library

1. Download a copy of the Fluid Infusion component library from:
    - http://fluidproject.org/products/fluid-infusion/download-infusion/
      You only really need the "Minified deployment package," but if you want to actually look at the code, you should download the "Source package."
2. Unpack the zip file you just downloaded, and place the resulting folder somewhere convenient for your development purposes.
   The folder will have the release number in its name (e.g. infusion-1.4/). The rest of this tutorial will use infusion-1.4 in its examples, but if you downloaded a different version, you'll have to adjust.

Step 1: Prepare your markup

The Rendered version of the Pager component works with the raw data that you want paged, and lays is out according to a template you provide in your mark-up.

The Pager is made up of several parts:

- a Pager Bar: The set of links to different pages (some people use two of these, one above the data, one below)
- the paged data itself
- a summary, such as "Items 20-30 of 76" (this is optional)
- a control to allow the user to change the number of items per page (this is also optional)

Let's say you want to display your roster results in a `<table>` element, and you want the page links at the top. We'll start simple, with just a top Pager Bar above the paged data (we'll add a summary and other enhancements later). Because you're dealing with thousands of student, you obviously don't want to have to manually create a `<table>` with thousands of rows – this is where the Pager's use of the Renderer comes in. In your HTML, you simply create a single table row that acts as a template to be used for each row. The same applies to the page links in the Pager Bar: you only define the minimum set of links necessary to illustrate how you want the links rendered.

So we might have something like this:

```
<div>
    <div>
        <ul class="demo-pager-bar fl-force-left">
            <li class="demo-pager-previous"><a href="#" title="Previous Page">&lt; Previous</a></li>
            <li>
                <ul class="demo-pager-links">
                    <!-- This is the template for a page link -->
                    <li class="demo-pager-pageLink-default"><a href="#">1</a></li>
                    <li class="demo-pager-pageLink-skip">...</li>
                    <li><a href="#">2</a></li>
                </ul>
            </li>
            <li class="demo-pager-next"><a href="#" title="Next Page">Next &gt;</a></li>
        </ul>
    </div>
    <div id="membership-table" class="fl-container-flex demo-pager-table-data">
       <table summary="Table of all students and teachers.">
            <thead>
                <tr>
                    <th class="demo-pager-head-members">Members</th>
                    <th class="demo-pager-head-email">Email</th>
                    <th class="demo-pager-head-role">Role</th>
                    <th><span class="demo-pager-head-comments">Comments</span></th>
                </tr>
            </thead>

            <tbody>
                <!-- This is the template for each data row -->
                <tr>
                    <td>Edee</td>
                    <td>edee@none.none</td>
                    <td>Instructor</td>
                    <td>The best teacher.</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Some notes about this mark-up:

- This code uses the Fluid Skinning System (FSS) for some layout, in particular, the "fl-force-left" and "fl-container-flex" classes.
- This code assumes you have a CSS file containing the various "demo-..." CSS classes for styling the markup.

Selectors

Next, you will need to let the Pager know which pieces of mark-up are to be used for what. This is accomplished though an `id` or `class` on an element. In particular, you'll need to identify

1. the main container of the entire Pager markup
2. the container of the Pager Bar
3. the different kinds of links in the Pager Bar
4. the table you're using for the data
5. the row template, and the columns within it

The Pager defines some default class names for some of these things. Others, you will have to define yourself. So:

1. the main container of the entire Pager markup: There is no default for this, so let's attach an `id` of "demo-pager-container" to our main `<div>`.
2. the container of the Pager Bar: The default for this is a class name of "flc-pager-top" so we'll add that class to the `<div>` containing our list of links (note that all default selectors for Fluid components start with "flc-", which is short for "fluid component." These selectors are used for manipulating the DOM only, and should not be used for styling.)
3. the different kinds of links in the Pager Bar: The pager defines a number of default class names for these links, so we'll use those:
   - "flc-pager-previous" for the link to the 'previous' page of data
   - "flc-pager-next" for the link to the 'next' page of data
   - "flc-pager-links" for the container for the page-specific links
   - "flc-pager-pageLink" for an individual page-specific link
   - "flc-pager-pageLink-skip" for the markup to use where there are gaps in the page links, used with the "gapped" page strategy. This is useful when there will be many links, and can be used to produce lists like "1 2 3 ... 7 8 9 ... 12 13")
4. the table you're using for the data: There is no default for this, so let's attach an `id` of "membership-table" to the `<table>` element.
5. the row template, and the columns within it: The pager requires a special `id` with a namespace of `rsf` for these items, so we'll define the namespace for the table, and add ids to the row and each of the column cells. Note that the row id must have a colon (:) at the end of it, to indicate that it will be repeated.

```
<div class="demo-pager-container">
    <div class="flc-pager-top">
        <ul class="demo-pager-bar fl-force-left">
            <li class="flc-pager-previous demo-pager-previous"><a href="#" title="Previous Page">&lt; Previous<
/a></li>
            <li>
                <ul class="flc-pager-links demo-pager-links">
                    <!-- This is the template for a page link -->
                    <li class="flc-pager-pageLink demo-pager-pageLink-default"><a href="#">1</a></li>
                    <li class="flc-pager-pageLink-skip demo-pager-pageLink-skip ">...</li>
                    <li class="flc-pager-pageLink"><a href="#">2</a></li>
                </ul>
            </li>
            <li class="demo-pager-next flc-pager-next"><a href="#" title="Next Page">Next &gt;</a></li>
        </ul>
    </div>
    <div id="membership-table" class="fl-container-flex demo-pager-table-data fl-pager-data">
       <table summary="Table of all students and teachers." xmlns:rsf="http://ponder.org.uk">
            <thead>
                <tr>
                    <th class="demo-pager-head-members">Members</th>
                    <th class="demo-pager-head-email">Email</th>
                    <th class="demo-pager-head-role">Role</th>
                    <th><span class="demo-pager-head-comments">Comments</span></th>
                </tr>
            </thead>

            <tbody>
                <!-- This is the template for each data row -->
                <tr rsf:id="row:">
                    <td><span rsf:id="user-link">Edee</span></td>
                    <td><span rsf:id="user-email">edee@none.none</span></td>
                    <td><span rsf:id="user-role">Instructor</span></td>
                    <td><span rsf:id="user-comment">The best teacher.</span></td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

This should get us started.

Step 2: Write the script, setting the options

You'll need to create a file, say `pagersetup.js`, to contain your initialization script - the script you write to apply the Pager to your roster:

```
jQuery(document).ready(function () {
    var opts = {
        // we'll go into this in a moment
    };
    fluid.pager("#demo-pager-container", opts);
});
```

Initializing a Pager is pretty simple: one call to the creator function, passing

- the selector for your main container (`"#demo-pager-container"`), and
- options for configuring your Pager. It's setting up the options where the magic happens.

The Pager supports a number of options that can be set to configure the behaviour of the component. The options argument to the creator function is a JavaScript object containing the various options, some of which may be objects themselves. Let's go over the options we'll need to set for our Pager (for detailed information about all supported options, see the Pager API page).

`bodyRenderer`

The `bodyRenderer` option allows you to specify and configure the subcomponent used to render the actual data. By default, no subcomponent is used (valid in the case where all of the necessary mark-up has been provided), but we want the Pager to render our data for us, so we will configure it to use the `fluid.pager.selfRender` subcomponent provided with the Pager.

The option value itself is an object containing the name of the subcomponent and configuration options for it:

```
var opts = {
    bodyRenderer: {
        type: "fluid.pager.selfRender",
        options: {
            selectors: {
                root: ".demo-pager-table-data"
            }
        }
    }
};
```

The `selectors` option to the subcomponent specifies the `<div>` that contains our data table.

`pagerBar`

The Pager will automatically create a Pager Bar component, but the `pagerBar` option allows you to customize some of the behaviour of the Pager Bar.

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {
        type: "fluid.pager.pagerBar",
        options: {
            pageList: {
                type: "fluid.pager.renderedPageList"
            }
        }
    }
};
```

The Pager Bar includes two subcomponents: PageList, which controls the actual list of page links, and PreviousNext, which controls the 'next' and 'previous' buttons (see Pager Subcomponents for more information). The default 'next'/'previous' subcomponent should be fine for us, but the default PageList is called a "direct" page list. This is only useful if you already have markup for each page link. Since we don't know how many pages we'll have, we need to use the "rendered" page list provided by the Pager.

`dataModel`

The `dataModel` option contains the actual data that you want paged, in the form of a pure-date JavaScript object. The structure of the object can take any form: paths expressed in component trees and configuration are expressed relative to this model.

For our scenario, our `dataModel` will look like this:

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {...},
    dataModel: {
        "entityPrefix": "membership",
        "membership_collection": [
            {
                "memberRole": "Instructor",
                "userDisplayName": "Joe I. Instructor",
                "userEmail": "joe@none.none",
                "userComment": "Needs improvement."
            },
            {
                "memberRole": "TA",
                "userDisplayName": "Mike A. Smith",
                "userEmail": "msmith@none.none",
                "userComment": " "
            },
            {
                "memberRole": "Student",
                "userDisplayName": "Jane Doe",
                "userEmail": "j.doe@none.none",
                "userComment": " "
            },
            ...
    }
};
```

While this snippet shows the data model being defined directly in the code, it is more likely that the actual data will be loaded through some other call to the server, such as fluid.fetchResources.

`dataOffset`

The `dataOffset` option tells the pager where in the `dataModel` to look for the actual data. This option is not always necessary, depending on the format of `dataModel`. In our case, we need to specify it, as shown on the right.

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {...},
    dataModel: {...},
    dataOffset: "membership_collection"
};
```

`model`

The `model` option contains the initial settings for the Pager model itself. This is different than the `dataModel`: `model` includes information such as the number of items which may be shown on the page, the sort order, the index of the current page, etc. The Pager includes defaults for these, but for our use case, we'll override one of the defaults: We'd like the initial number of items per page to be larger than the default of 10:

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {...},
    dataModel: {...},
    dataOffset: "membership_collection",
    model: {
        pageSize: 20
    }
};
```

For information about all of the model values that can be customized through the options, see Pager API.

`columnDefs`

The `columnDefs` option defines the rules for extracting and presenting data from the data model into your columns. It is an array containing one entry for each column which is to be rendered in the table.

Our data model is relatively simple, so our `columnDefs` entries only need to specify two things:

- `key`: This is the `rsf:id` of the element in the markup that is the column where the data should go
- `valuebinding`: This is the path into your `dataModel` of the data that should be rendered into the column

If you recall, our template for the rows looked like this:

```
<tr rsf:id="row:">
     <td><span rsf:id="user-link">Edee</span></td>
     <td><span rsf:id="user-email">edee@none.none</span></td>
     <td><span rsf:id="user-role">Instructor</span></td>
     <td><span rsf:id="user-comment">The best teacher.</span></td>
</tr>
```

and our data model for the roster looked like this:

```
{
    "memberRole": "Instructor",
    "userDisplayName": "Joe I. Instructor",
    "userEmail": "joe@none.none",
    "userComment": "Needs improvement."
},
```

so the `columnDefs` structure will be as shown on the right.

```
var rosterColumnDefs = [
    {
        key: "user-link",
        valuebinding: "*.userDisplayName"
    },
    {
        key: "user-email",
        valuebinding: "*.userEmail"
    },
    {
        key: "user-role",
        valuebinding: "*.memberRole"
    },
    {
        key: "user-comment",
        valuebinding: "*.userComment"
    }
];
var opts = {
    bodyRenderer: {...},
    pagerBar: {...},
    dataModel: {...},
    dataOffset: "membership_collection",
    columnDefs: rosterColumnDefs
};
```

These options are the minimum you will need to set to use the rendered Pager.

Step 3: Add the script and styles to your HTML

You'll need to add your initialization script, along with the Fluid library and Pager style sheets, to you HTML file, as shown below::

```
<link rel="stylesheet" type="text/css" href="framework/fss/css/fss-reset.css" />
<link rel="stylesheet" type="text/css" href="framework/fss/css/fss-layout.css" />
<link rel="stylesheet" type="text/css" href="components/pager/css/Pager.css" media="all" />
<link rel="stylesheet" type="text/css" href="lib/jquery/plugins/tooltip/css/jquery.tooltip.css" media="all" />
<link rel="stylesheet" type="text/css" href="lib/jquery/ui/css/jquery.ui.theme.css" />

<script type="text/javascript" src="infusion-1.3/InfusionAll.js"></script>
<script type="text/javascript" src="pagerSetup.js"></script>
```

NOTE that the `InfusionAll.js` file is minified - all of the whitespace has been removed, so it isn't really human-readable. If you're using the source distribution and you want to be able to debug the code, you'll want to include each of the required files individually. This would look like this:

```
<!-- Stylesheets -->
<link rel="stylesheet" type="text/css" href="framework/fss/css/fss-reset.css" />
<link rel="stylesheet" type="text/css" href="framework/fss/css/fss-layout.css" />
<link rel="stylesheet" type="text/css" href="components/pager/css/Pager.css" media="all" />
<link rel="stylesheet" type="text/css" href="lib/jquery/plugins/tooltip/css/jquery.tooltip.css" media="all" />
<link rel="stylesheet" type="text/css" href="lib/jquery/ui/css/jquery.ui.theme.css" />

<!-- Scripts -->
<script type="text/javascript" src="lib/jquery/core/js/jquery.js"></script>
<script type="text/javascript" src="lib/jquery/ui/js/jquery.ui.core.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="lib/jquery/ui/js/jquery.ui.widget.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="lib/jquery/ui/js/jquery.ui.position.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="lib/jquery/plugins/bgiframe/js/jquery.bgiframe.js"></script>   <!-- New in
v1.3 -->
<script type="text/javascript" src="lib/jquery/plugins/tooltip/js/jquery.ui.tooltip.js"></script>   <!-- New in
v1.3 -->
<script type="text/javascript" src="lib/json/js/json2.js"></script>   <!-- New in v1.3 -->

<script type="text/javascript" src="framework/core/js/Fluid.js"></script>
<script type="text/javascript" src="framework/core/js/FluidDOMUtilities.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="framework/core/js/FluidDocument.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="framework/core/js/jquery.keyboard-a11y.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="framework/core/js/DataBinding.js"></script>
<script type="text/javascript" src="framework/core/js/FluidRequests.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="lib/fastXmlPull/js/fastXmlPull.js"></script>
<script type="text/javascript" src="framework/renderer/js/fluidParser.js"></script>
<script type="text/javascript" src="framework/renderer/js/fluidRenderer.js"></script>

<script type="text/javascript" src="components/tooltip/js/Tooltip.js"></script>   <!-- New in v1.3 -->
<script type="text/javascript" src="components/pager/js/Pager.js"></script>
```

But all of these individual JavaScript files are not necessary to make it work - the `InfusionAll.js` file has everything you need.

Other Customizations

Paging Strategy

The Rendered page list we selected for our Pager Bar renders one link for every page of data. This is fine if you have a handful of pages, but what if you have tens of page? or hundreds? The rendered page list uses a strategy for rendering the list of links. The default is a "direct list" – one link per page. For cases where you expect to have many, many pages, the Pager provides a strategy called the "Gapped Page Strategy." This strategy displays the links for first few pages and the last few pages, and for some pages around the current page, but simply shows a gap for any other pages. As different pages are selected, the gaps are automatically adjusted.

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {
        type: "fluid.pager.pagerBar",
        options: {
            pageList: {
                type: "fluid.pager.renderedPageList",
                options: {
                    pageStrategy: fluid.pager.gappedPageStrategy(3, 1)
                }
            }
        }
    }
};
```

We construct the gapped page strategy by calling the function `fluid.pager.gappedPageStrategy(locality, midLocality)`. We will configure our PageList to only include the first and last 3 pages (`locality`), or within 1 of the current page position (`midLocality`).

Sortable Columns

The Pager provides the ability to sort the table's rows with respect to a column. This is accomplished very easily, by simply

1. adding a flag to the `columnDefs` information, and
2. ensuring that the headers of the sortable columns are links.

```
var rosterColumnDefs = [
    {
        key: "user-link",
        valuebinding: "*.userDisplayName",
        sortable: true
    },
    {
        key: "user-email",
        valuebinding: "*.userEmail",
        sortable: true
    },
    {
        key: "user-role",
        valuebinding: "*.memberRole",
        sortable: true
    },
    {
        key: "user-comment",
        valuebinding: "*.userComment",
        sortable: false
    }
];
```

This example defines the name, email and role columns as sortable, and the comments column as not sortable. (Note that not sortable is the default, so strictly speaking, it is not necessary to set `sortable` to `false`).

```
...
<table summary="Table of all students and teachers." xmlns:rsf="http://ponder.org.uk">
    <thead>
        <tr rsf:id="header:">
            <th class="demo-pager-head-members"><a rsf:id="user-link" href="#">Members</a></th>
            <th class="demo-pager-head-email"><a rsf:id="user-email" href="#">Email</a></th>
            <th class="demo-pager-head-role"><a rsf:id="user-role" href="#">Role</a></th>
            <th><span class="demo-pager-head-comments">Comments</span></th>
        </tr>
    </thead>
    <tbody>
    ...
```

When we convert the header text to a link, we must assign the link the same `rsf:id` as we did for the data in the columns themselves. This ensures that when the header is used to sort the data, the correct column of data is used.

Range Annotation

Another feature the Pager offers is the ability to provide information about what range of data is associated with a given page. This is an annotation attached to each page link, displayed as a tooltip when a page link is hovered over and spoken aloud by screen readers when a page link receives keyboard focus.

To enable the range annotation, use the `annotateColumnRange` option of the Pager. Set its value to the `rsf:id` of the column to extract the range data from:

```
var opts = {
    bodyRenderer: {...},
    pagerBar: {...},
    dataModel: {...},
    dataOffset: "membership_collection",
    columnDefs: rosterColumnDefs
    annotateColumnRange: "user-link"
};
```

Here, we're selecting the user name column as the source data for the range annotation. With this option enabled, the range annotation tooltip will display the first and last user names that would appear on the given page's data.