04. DATATABLES Data Renderers.

4.1. Renderers

There are occasions when working with tables that the data source for rows in the table do not contain the value that you wish to show directly in the table. You may wish to transform it to a different representation (*a time stamp into a human readable format*), combine data points (*first and last names*) or perform some computation on the value (*calculating margin from turnover and expense values*).

This transformation of the original data into the value that will be shown in the DataTable is called **rendering** in DataTables' terminology and is performed using the **columns.render** option.

4.2. Data Rendering

The primary advantage of using a data renderer in DataTables is that **you can modify the output data without modifying the original data**. The **columns.data** method can be used to get and set data, but the set operation adds significant complexity and it is recommended that the **columns.data** option be used simply to point to the original representation of the data and allow a renderer (*columns.render*), which is read only, to transform the data.

columns.render can be utilised in a number of different ways:

- As a **function** to transform data
- As a **string** to select data from an object

Functions

Using **columns.render** is the most common method as it provides absolute control over the data that will be displayed to the end user (*this is a regular Javascript function, so you can do virtually anything you wish with the data*).

The function is passed in three parameters:

- The **data** that is pointed to by **columns.data**. If columns.data is null, null will be the value given here.
- The **data type** being requested by DataTables - this allows the function to support orthogonal data.
- The original and **full data object** or array for the row.

The value that is returned from the function is whatever DataTables will use for the data being requested (*display, ordering, search, etc*).

Consider for example the following data structure which contains the data for a row:

Example:

```
{
    "product": "Toy car",
    "creator": {
        "firstName": "Fiona",
        "lastName": "White"
    },
    "created": "2015-11-01",
    "price": 19.99,
    "cost": 12.53
}
```

Adding formatting

In our DataTable, if we wish to have a column that **shows the price**, it is relatively common to wish to **prefix it with a currency sign**. In this case we use an euro sign € (*see also the built-in number renderer below which provides advanced formatting options*):

Example:

```
{
    data: 'price',
    render: function ( data, type, row ) {
        return '€'+ data;
    }
}
```

Joining strings

In our DataTable if we wish to have a single column that shows the full name of the creator we can concatenate strings using the following columns definition (note in particular how the create object is passed in as the first parameter due to its assignment using the **columns.data** option):

Example:

```
{
    data: 'creator',
    render: function ( data, type, row ) {
        return data.firstName +' '+ data.lastName;
        }
    }
```

Transforming data

For another column we wish to display the created value, but formatted using the US standard MM-DD-YYYY formatting. This can be done simply by splitting the string and rearranging the component parts. We also wish for the date to be sortable, and since DataTables has built in support for ISO8601 formatted string (the original format), we wish to perform the transformation only for the display and filter data type - see orthogonal data for more):

Example:

```
    {
        data: 'created',
        render: function ( data, type, row ) {
            var dateSplit = data.split('-');
            return type === "display" || type === "filter" ?
                dateSplit[1] +'-'+ dateSplit[2] +'-'+ dateSplit[0] :
                data;
        }
    }
```

Computing values

Finally, to create a margin column from the price and cost fields we can use a function to compute the **required values** - note that in this case **columns.data** is **null** - as a result the first parameter passed into the **columns.render** method is also **null**, but the third parameter provides access to the original data source object, so we can continue use the data from there:

Example:

```
    {
        data: null,
        render: function ( data, type, row ) {
            return Math.round( ( row.price - row.cost ) / row.price * 100 )+'%';
        }
    }
```

Strings

A less common option for formatters is as a string to simply point at the data that should be used in the table. This is similar to the way that **columns.data** is often used, although keep in mind that the renderer will only have access to the data pointed to by **columns.data** rather than the full row.

Continuing the examples using the JSON data structure from above, consider a column that should show the first name of the creator:

Example:

```
{
    data: 'creator',
    render: 'firstName'
}
```

There is no advantage to this method over simply using data: '*creator.firstName*' in the example presented here, but if you have complex data with orthogonal data included in the data source object, this can sometimes be useful.

4.3. Built-in helpers

DataTables has two built in rendering helpers that can be used to easily format data - more can be added using plug-ins (see below):

- **number** - for formatting numbers
- **text** - to securely display text from a potentially unsafe source (*HTML entities are escaped*).

The built in rendering helpers can be accessed under the object **$.fn.dataTable.render**. They are functions (allowing options to be passed into them) which should be immediately executed and their result assigned to the **columns.render** method. This might sound a little complicated, but it simply means you would use something like the following:

Example:

```
{
    data: 'price',
    render: $.fn.dataTable.render.number( ... )
}
```

Number helper

The number helper provides the ability to easily format, you guessed it, numbers! When dealing with numbers, you may often wish to add formatting such as prefix and postfix characters (currency indicators for example), use a thousands separator and specify a precision for the number. This is all possible with the number helper.

The helper function takes up to five optional parameters:

- **The thousands separator** (required)
- **Decimal separator** (required)
- **Floating point precision** - 0 for integers, 1 for a single decimal place, etc (optional)
- **A prefix string** (optional)
- **A postfix string** (optional).

For example, to display the price data point from the data structure shown above in the format $19.99 we would use:

Example:

```
{
    data: 'price',
    render: $.fn.dataTable.render.number( ',', '.', 2, '$' )
}
```

This example doesn't require the thousands separator, but for larger values such as 1000 they would be formatted as $1,000.00.

Note that if the number helper encounters a value which is not a valid number (*either number or string that contains a number*) it will return the value after escaping any HTML entities in it (*to help protect against potential security attacks*).

Text helper

The text helper will ensure that any potentially dangerous HTML in the source data will not be executed by escaping the HTML entities. This can be useful if the data being loaded may come from a potentially untrusted data source and can help mitigate XSS attacks.

The text helper doesn't take any parameters making its use simply:

Example:

```
{
    data: 'product',
    render: $.fn.dataTable.render.text()
}
```

4.4. Custom Helpers

Rendering helpers are simply functions which are attached to **$.fn.dataTable.render** to make them easily accessible from a single location. These functions must return a function that will operate with the **columns.rende**r method.

Consider for example the following simple plug-in that will truncate text after a given number of characters and show ellipsis if the string is longer that the number of characters allowed:

Example:

```
$.fn.dataTable.render.ellipsis = function ( cutoff ) {
    return function ( data, type, row ) {
        if ( type === 'display' ) {
            var str = data.toString(); // cast numbers

            return str.length < cutoff ?
                str :
                str.substr(0, cutoff-1) +'&#8230;';
        }

        // Search, order and type can use the original data
        return data;
    };
};
```

We can then use that in our DataTables column definitions:

Example:

```
{
    data: 'description',
    render: $.fn.dataTable.render.ellipsis( 10 )
}
```

A more comprehensive ellipsis rendering helper is available in the DataTables plug-ins repository with word break and HTML escaping control. Others will be available in the same repository as they are developed.

https://github.com/DataTables/Plugins/tree/master/dataRender