

# DYNATABLE

## HTML5+JSON interactive table plugin.

Dynatable is a funner, semantic, interactive table plugin using jQuery, HTML5, and JSON. And it's not just for tables.



**Download**

(<http://jspkg.com/package/jquery-dynatable>)



**View Source**

(<https://github.com/alfajango/dynatable>)

31,491 downloads

Star

2,201

Fork

223

### Demo

How it works

Normalization

Converting attribute names

Existing JSON

JSON from AJAX

Lists and non-Tables

Operations

Sorting

Querying

Paginating

Record Count

PushState

Processing Indicator

Rendering

A Stylized List

An Interactive Chart

Configuration

Data Attributes

Event Hooks

API

dom

domColumns

records

recordsCount

processingIndicator

state

sorts

sortsHeaders

queries



inputSearch

paginationPage

paginationPerPage

paginationLinks

Project Resources

share this:  

Latest Update on Github | Sep 14, 2014 **Commit ce207a** Added link in contributi...

(<https://github.com/alfajango/jquery-dynatable/commit/ce207ab5070bf6bf5482d9b4178b2b1b7cc88c98>)

## Demo

Show: 10

Search:

Rank	Country ▼	US \$	Year
158	 Zimbabwe ( <a href="http://en.wikipedia.org/wiki/Zimbabwe">http://en.wikipedia.org/wiki/Zimbabwe</a> )	741	2011
139	 Zambia ( <a href="http://en.wikipedia.org/wiki/Zambia">http://en.wikipedia.org/wiki/Zambia</a> )	1,414	2011
142	 Yemen ( <a href="http://en.wikipedia.org/wiki/Yemen">http://en.wikipedia.org/wiki/Yemen</a> )	1,340	2011
-	World ( <a href="http://en.wikipedia.org/wiki/World">http://en.wikipedia.org/wiki/World</a> ) <sup>[8]</sup>	10,144	2011
141	 Vietnam ( <a href="http://en.wikipedia.org/wiki/Vietnam">http://en.wikipedia.org/wiki/Vietnam</a> )	1,374	2011
60	 Venezuela ( <a href="http://en.wikipedia.org/wiki/Venezuela">http://en.wikipedia.org/wiki/Venezuela</a> )	10,610	2011
120	 Vanuatu ( <a href="http://en.wikipedia.org/wiki/Vanuatu">http://en.wikipedia.org/wiki/Vanuatu</a> )	3,036	2011
133	 Uzbekistan ( <a href="http://en.wikipedia.org/wiki/Uzbekistan">http://en.wikipedia.org/wiki/Uzbekistan</a> )	1,572	2011
48	 Uruguay ( <a href="http://en.wikipedia.org/wiki/Uruguay">http://en.wikipedia.org/wiki/Uruguay</a> )	13,914	2011
14	 United States ( <a href="http://en.wikipedia.org/wiki/United_States">http://en.wikipedia.org/wiki/United_States</a> )	48,387	2011

Showing 1 to 10 of 186 records

Pages: Previous 1 2 3 ... 19 Next

\* List of countries by GDP per capita from Wikipedia

([http://en.wikipedia.org/wiki/List\\_of\\_countries\\_by\\_GDP\\_\(nominal\)\\_per\\_capita](http://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)_per_capita))

To get started, simply install jquery.dynatable.js (along with jQuery), and add the following in the `document.ready` or after the table:

```
$('#my-table').dynatable();
```

# How it works

Dynatable does three things:

Name	Period	Location	Diet
Antarctopelta	Cretaceous	Antarctica	herbivore
Atlascopcosaurus	Cretaceous	Australia	herbivore
Australovenator	Cretaceous	Australia	carnivore
Austrosaurus	Cretaceous	Australia	herbivore



Name ▲	Period ▼	Location	Diet ▼
Antarctopelta	Cretaceous	Antarctica	herbivore
Atlascopcosaurus	Cretaceous	Australia	herbivore
Australovenator	Cretaceous	Australia	carnivore
Austrosaurus	Cretaceous	Australia	herbivore

## 1. Read / Normalize

The HTML table is scanned and normalized into an array of JSON objects (or collection) where each JSON object (or record) corresponds to a row in the table.

## 2. Operate

The JSON collection can be sorted, searched/filtered, and paginated/sliced.

## 3. Write / Render

The results of the Operate step are rendered back to the DOM in the body of the table.

This 3-step approach has several advantages:

### • Efficient reading/operating/writing

Since the logic and operations occur on the JSON collection, the DOM operations (reading and writing/drawing) are grouped together, making interactions quick and efficient.

### • Operations are simple JavaScript

An operation is simply a function that acts on the normalized JSON collection; sorting, filtering, and paginating are straight forward in JavaScript.

The built-in functions are easy to augment with your own custom sorting and querying functions.

### • Steps can be customized, swapped or skipped

Since the normalization, operation, and rendering modules are separated, each can easily be customized, replaced, or skipped.

Already have a JSON API to work with? Skip the Read step. Want to add paginating, filtering, and sorting to a chart? Customize the Render step.

# Normalization

The first module normalizes an HTML table into a JSON collection. Dynatable names the attributes of each record according to the table heading, so that the JSON collection is human-readable and easy to work with.

The following table:

Results in this JSON collection:

```
<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Hobby</th>
      <th>Favorite Music</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Fred</td>
      <td>Roller Skating</td>
      <td>Disco</td>
    </tr>
    <tr>
      <td>Helen</td>
      <td>Rock Climbing</td>
      <td>Alternative</td>
    </tr>
    <tr>
      <td>Glen</td>
      <td>Traveling</td>
      <td>Classical</td>
    </tr>
  </tbody>
</table>
```

```
[
  {
    "name": "Fred",
    "hobby": "Roller Skating",
    "favoriteMusic": "Disco"
  },
  {
    "name": "Helen",
    "hobby": "Rock Climbing",
    "favoriteMusic": "Alternative"
  },
  {
    "name": "Glen",
    "hobby": "Traveling",
    "favoriteMusic": "Classical"
  }
]
```

## Converting attribute names

By default, dynatable converts headings to JSON attribute names using:

Style	Example
<code>camelCase</code> (default)	<code>favoriteMusic</code>
<code>trimDash</code>	<code>Favorite-Music</code>
<code>dashed</code>	<code>favorite-music</code>
<code>underscore</code>	<code>favorite_music</code>
<code>lowercase</code>	<code>favorite music</code>

```
$('#my-table').dynatable({
  table: {
    defaultColumnIdStyle: 'trimDash'
  }
});
```

```
$.dynatableSetup({
  table: {
    defaultColumnIdStyle:
'underscore'
  }
});
```

PROTIP: When using dynatable in a Rails application, set the global style to `underscore`, matching the Rails parameter and input field naming conventions. This is useful when getting the JSON data via AJAX from Rails, or when connecting dynatable events with form inputs on the page).

We could also define our own column-name transformation function. Consider the following table with these column headings:

Name	Hobby	Favorite Music
Fred	Roller Skating	Disco
Helen	Rock Climbing	Alternative
Glen	Traveling	Classical

We can set up our own function for transforming column labels to JSON property names of our desired custom format when we instantiate dynatable on the table above:

```
$('#text-transform-example').bind('dynatable:preinit', function(e,
dynatable) {
  dynatable.utility.textTransform.myNewStyle = function(text) {
    return text
      .replace(/\s+/, '_')
      .replace(/[A-Z]/, function($1){ return $1 + $1 });
  };
}).dynatable({
  table: {
    defaultColumnIdStyle: 'myNewStyle'
  },
  features: {
    paginate: false,
    search: false,
    recordCount: false,
    perPageSelect: false
  }
});
```

You may edit the code to the left to experiment with different custom text-transform functions.

Click the button to the right.  
Do it!

Run Code

Click this button to run the code above, populating the box on the left with the resulting JSON collection built by dynatable.

Sometimes, we need columns with labels different than the record attribute name. If a column heading contains the `data-dynatable-column` attribute, the associated record attribute will be named by that value.

So this:

Would result in:

```
<table id="my-final-table">
  <thead>
    <th data-dynatable-column="name">Band</th>
    <th>Hit</th>
  </thead>
  <tbody>
    ...
  </tbody>
</table>
```

```
[
  {
    "name": ...,
    "song": ...
  },
  {
    "name": ...,
    "song": ...
  }
]
```

The default behavior makes it easy to make an existing HTML table dynamic. But we're not limited to reading tables.

## Existing JSON

Perhaps we already have our data in JSON format. We can skip the initial record normalization by setting up an empty table for rendering and directly passing our data into dynatable:

HTML table to render records:

This is a `pre#json-records` element:

```
<table id="my-final-table">
  <thead>
    <th>Band</th>
    <th>Song</th>
  </thead>
  <tbody>
  </tbody>
</table>
```

```
[
  {
    "band": "Weezer",
    "song": "El Scorcho"
  },
  {
    "band": "Chevelle",
    "song": "Family System"
  }
]
```

Of course we could just code the json data directly in our JavaScript on the right, but what's the fun in that? As a bonus, edit the JSON data to the right and watch the data in the table update in real-time. →

```
var $records = $('#json-records'),
    myRecords = JSON.parse($records.text());
$('#my-final-table').dynatable({
  dataset: {
    records: myRecords
  }
});
```

Show: 10 ▼

Search:

Band	Song
Weezer	El Scorcho
Chevelle	Family System

Showing 2 of 2 records

Pages: Previous 1 Next

## JSON from AJAX

Or maybe, we want to fetch the data via AJAX:

```
<table id="my-ajax-table">
  <thead>
    <th>Some Attribute</th>
    <th>Some Other Attribute</th>
  </thead>
  <tbody>
    </tbody>
</table>
```

```
$('#my-ajax-table').dynatable({
  dataset: {
    ajax: true,
    ajaxUrl: '/dynatable-ajax.json',
    ajaxOnLoad: true,
    records: []
  }
});
```

View AJAX data (/dynatable-ajax.json)

NOTE: When using AJAX to load data, operations such as sorting, searching, and paginating are performed on the server before building the returned JSON. This example has these features disabled since, we're just loading a static JSON file for the purposes of documentation.

Some Attribute	Some Other Attribute
I am record one	Fetchd by AJAX
I am record two	Cuz it's awesome
I am record three	Yup, still AJAX

Showing 3 records

When using Dynatable in "AJAX mode" ( `dataset.ajax = true` ), delegates all operations (pagination, sorting, and querying/filtering) to the server. For each operation, dynatable culls the parameters (sort, search, page) into an AJAX request and fetches the results from `dataset.ajaxUrl` (if this setting isn't set, it will send an AJAX request to the URL of the current page).

AJAX mode is intended to be used when you want the server to look up the records only as needed. This generally means your server is looking up the records from a database using the database's query, limit, and offset functions to select the appropriate subset of records.

Because your server is only ever returning a subset of the records at a time to Dynatable, the response must contain some extra meta-data. The following format is the default format expected by Dynatable.

```
{
  "records": [
    {
      "someAttribute": "I am record one",
      "someOtherAttribute": "Fetchd by AJAX"
    },
    {
      "someAttribute": "I am record two",
      "someOtherAttribute": "Cuz it's awesome"
    },
    {
      "someAttribute": "I am record three",
      "someOtherAttribute": "Yup, still AJAX"
    }
  ],
  "queryRecordCount": 3,
  "totalRecordCount": 3
}
```

If you instead want to fetch all records from the server at once via AJAX, you may consider leaving AJAX mode off, fetching the records, and calling Dynatable with the normal JSON recordset returned by the server:

```
$.ajax({
  url: 'ajax_data.json',
  success: function(data){
    $('#my-final-table').dynatable({
      dataset: {
        records: data
      }
    });
  }
});
```

## Lists and non-Tables

Or maybe we do need the normalization step, but we want to read the data from an unordered list instead of a table:

We can use the `table` settings to configure such awesomeness. We'll use the `table.bodyRowSelector` setting to tell dynatable to use `li` elements as record rows instead of the default `tr` elements, and we'll use the `writers._rowWriter` setting to tell dynatable how to process each `li` into a JSON record object.

Dynatable will call the `readers._rowReader` function once for each record in the `table.bodyRowSelector` collection, and pass it the current count index, the DOM element, and the JSON record. This allows full control over which data in the DOM maps to which data in the JSON:

*NOTE: We'll also need a `readers._rowWriter` function to tell dynatable how to write the JSON records back to the page, but we'll get to that in the Render section.*

The following HTML:

Will result in the following JSON:

```
<ul id="my-list">
  <li>
    <span class="name">Fender Custom Esquire GT</span>
    <span class="type">Guitar</span>
    <span class="price">450.00</span>
  </li>
  <li>
    <span class="name">ESP LTD B4-E</span>
    <span class="type">Bass</span>
    <span class="price">400.00</span>
  </li>
</ul>
```

```
[
  {
    "name": "Fender Custom Esquire GT",
    "type": "Guitar",
    "price": 450.0
  },
  {
    "name": "ESP LTD B4-E",
    "type": "Bass",
    "price": 400.0
  }
]
```

And JavaScript:

```
$('#my-list').dynatable({
  table: {
    bodyRowSelector: 'li',
    rowReader: function(index, li, record) {
      var $li = $(li);
      record.name = $li.find('.name').text();
      record.type = $li.find('.type').text();
      record.price =
parseFloat($li.find('.price').text());
    }
  }
});
```

## Operations

Once we have our JSON dataset, we can perform all our interactive and dynamic logic directly on the JSON using JavaScript. By default, dynatable comes with functions for sorting, filtering (aka searching), and paginating.

By default, dynatable performs all operations on the JSON record collection in the page. However, if `dataset.ajax` is enabled, dynatable simply passes the operations (pagination, queries, and sort columns) as parameters to the AJAX URL, thereby delegating the logic to your server-side code.

The parameter names for pushState and AJAX requests can be customized in the `params` configuration settings for dynatable.

## Sorting



Dynatable allows for single or multi-column, smart sorting out of the box.




Dynatable can be made aware of the value types of each column, or record property, so that e.g. dates and numbers are sorted properly (plain-text sorting would cause February to come before January, and 10 to come before 2). By default, if dynatable detects HTML code within the value of a record (such as an `img` tag, it will automatically sort and search based on the text-equivalent value of the cell, so sorting won't be affected by HTML tags or attributes).

### Basic Sorting

Click the header rows below to sort by each column. Click a header once for ascending, again for descending, and again to stop sorting by that column.

Hold shift and click a second row to add secondary sorting, and so on.

Make	Model	Year ▲	Price
 Ford	Escape	2001	4,000
 Mini	Cooper	2001	8,500

 Ford	Focus SVT	2003	9,000
 Volkswagen	Jetta Wolfsburg	2008	11,000
 Ford	Focus	2013	20,000

In the example above, we run the "Price" column values through an "reader" function which returns a JavaScript `Number` and parses out the comma separator. Likewise, we then run it through a rendering "writer" which re-inserts the comma when rendering the number back to the DOM.

## Sort by Another Value

Sometimes, we need one column to sort based on some other attribute. For example, maybe we have a column which needs to sort on another hidden column. We can use the `data-dynatable-sorts` attribute on the column header to let dynatable know.







```
<table id="sorting-example">
  <thead>
    <tr>
      <th>Name</th>
      <th data-dynatable-sorts="computerYear">Year</th>
      <th style="display: none">Computer Year</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Steve</td>
      <td>Two Thousand and Thirteen</td>
      <td>2013</td>
    </tr>
  </tbody>
</table>
```

In the above example, dynatable will detect that the last column heading is hidden, and will hide all cells under that column, and it will sort the "Year" column based on the attribute in the last column.

If we have a column we don't want to be sortable, we just add the `data-dynatable-no-sort` attribute.

## Custom Sort Functions

We can also use our own custom sort function. This demo sorts the "color" column by the average color content in the images, from greenish to bluish to reddish (using javascript and canvas in our sorting function to evaluate the color content of each image):

Sort by Color	
	Cerasinops
	Ceratosaurus
	Allosaurus
	Tyrannosaurus
	Brachylophosaurus
	Albertaceratops

We may also sort programmatically with the dynatable API. For example, let's add a button which sorts our table by dinosaur names, and a button that clears all our sorts, putting the records back in their original order:

Sort A-Z

Clear Sorts

The code for the buttons above:

```
$('#sorting-function-example').bind('dynatable:init',
function(e, dynatable) {

  $('#sorting-function-example-button').click(
function(e) {
  // Clear any existing sorts
  dynatable.sorts.clear();
  dynatable.sorts.add('name', 1) // 1=ASCENDING,
-1=DESCENDING
  dynatable.process();
  e.preventDefault();
});

  $('#sorting-function-example-clear-button').click(
function(e) {
  dynatable.sorts.clear();
  dynatable.process();
  e.preventDefault()
});
});
```



*\* Images from List of North American dinosaurs from Wikipedia*  
([http://en.wikipedia.org/wiki/List\\_of\\_North\\_American\\_dinosaurs](http://en.wikipedia.org/wiki/List_of_North_American_dinosaurs))

There are a couple different ways to achieve the custom color sorting above, and it's useful to explore each way to gain a better understanding of what's possible.

### Creating a Custom Sort Function

The first way is to create a custom sort function, add it to dynatable's list of sort functions in `sorts.functions`, and then tell dynatable to use that function when sorting that column.

A sort function takes in the two records being compared (a and b below), the attribute column currently being sorted, and the direction (1 for ascending, -1 for descending). The function needs to return a positive number (if a is higher than b), a negative number (if b is higher than a), or 0 (if a and b are tied).

```
// Our custom sort function
function rgbSort(a, b, attr, direction) {

  // Assuming we've created a separate function
  // to get the average RGB value from an image.
  // (see source for example above for getAverageRGB function)
  var aRgb = getAverageRGB(a.img),
      bRgb = getAverageRGB(b.img),
      aDec = ( aRgb.r << 16 ) + ( aRgb.g << 8 ) + aRgb.b,
      bDec = ( bRgb.r << 16 ) + ( bRgb.g << 8 ) + bRgb.b,
      comparison = aDec - bDec;

  return direction > 0 ? comparison : -comparison;
};

// Wait until images are loaded
$(window).load(function() {
  $('#sorting-function-example')

    // Add our custom sort function to dynatable
    .bind('dynatable:init', function(e, dynatable) {
      dynatable.sorts.functions["rgb"] = rgbSort;
    })

    // Initialize dynatable
    .dynatable({
      features: {
        paginate: false,
        search: false,
        recordCount: false
      },
      dataset: {
        // When we sort on the color column,
        // use our custom sort added above.
        sortTypes: {
          color: 'rgbSort'
        }
      },
      readers: {
        color: function(cell, record) {
          var $cell = $(cell);

          // Store the average RGB image color value
          // as a decimal in "dec" attribute.
          record['img'] = $cell.find('img').get(0);

          // Return the HTML of the cell to be stored
          // as the "color" attribute.
          return $cell.html();
        }
      }
    });
});
```

The sort function gets run between each pair of records to determine which comes first. This means it gets run  $n!$  times (where n is the number of records), or  $n-1$  times for each record.



So, the average RGB values in this example are being re-computed multiple times for each record. This kills the efficiency.

### Creating a Custom Attribute to Sort On

Instead, it's much more efficient to compute values only once for each record and store them as record attributes. We were already storing the image file above for each record, so why not go ahead and store the RGB values too?

Furthermore, notice that in our custom `rgbSort` function above, after it calculates the RGB value for each record, it's just doing a standard number comparison (by subtracting one value from the other). Dynatable has built-in "number" sorting.

```
$(window).load(function() {
  $('#sorting-function-example')

  // Initialize dynatable
  .dynatable({
    features: {
      paginate: false,
      search: false,
      recordCount: false
    },

    // We have one column, but it contains multiple types of info.
    // So let's define a custom reader for that column to grab
    // all the extra info and store it in our normalized records.
    readers: {
      color: function(cell, record) {

        // Inspect the source of this example
        // to see the getAverageRGB function.
        var $cell = $(cell),
            rgb = getAverageRGB($cell.find('img').get(0)),
            dec = ( rgb.r << 16 ) + ( rgb.g << 8 ) + rgb.b;

        // Store the average RGB image color value
        // as a decimal in "dec" attribute.
        record['dec'] = dec;

        // Grab the dinosaur name.
        record['name'] = $cell.text();

        // Return the HTML of the cell to be stored
        // as the "color" attribute.
        return $cell.html();
      }
    }
  });
})
```

We could now create a custom sort function for the "color" column, to make sure it sorts based on the "dec" attribute instead. Or, we could just tell dynatable to sort the "color" column based on the "name" attribute directly in our table with `data-dynatable-sorts`:

```
<table>
  <thead>
    <tr>
      <th data-dynatable-column="color" data-dynatable-sorts="dec">Sort by Color</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td> Cerasinops</td>
    </tr>
    <!-- ... -->
  </tbody>
</table>
```

## Querying

(aka filtering or searching)

In addition to sorting, we can also query the data by some term or value. By default, dynatable includes a search box which matches from the plain-text values (case-insensitive) across all attributes of the records. Try it in the demo at the top of this page, by typing in the search box above the table and hitting "Enter" or "Tab".

### Custom Query Functions

Queries can also be added programmatically via JavaScript to be processed by dynatable. We simply add a query key-value to the `dataset.queries` array, where the key matches the JSON record attribute you'd like to match, and the value is what we're matching.

Below, we'll include the default text search, and additionally include our own "Year" filter.

```
<select id="search-year" name="year">
  <option></option>
  <option>2001</option>
  <option>2003</option>
  <option>2008</option>
  <option>2013</option>
</select>
```






NOTE: This JS is the long version, to show how customizable queries are. See below for the easier, built-in way to add your own query controls. →

```
var dynatable = $('#search-example').dynatable({
  features: {
    paginate: false,
    recordCount: false,
    sorting: false
  }
}).data('dynatable');

$('#search-year').change( function() {
  var value = $(this).val();
  if (value === "") {
    dynatable.queries.remove("year");
  } else {
    dynatable.queries.add("year",value);
  }
  dynatable.process();
});
```

Year:

Search:

Make	Model	Year
 Volkswagen	Jetta Wolfsburg	2008
 Ford	Focus	2013
 Ford	Escape	2001
 Mini	Cooper	2001
 Ford	Focus SVT	2003

There's a shortcut to the above code; to hook up our own search filters, we can just pass an array of jQuery selectors which point to our filter inputs. Instead of binding to our input's change event, adding the input's value to the queries array and calling the `dynatable.process()` function, we could have just done this:

```
$('#search-example').dynatable({
  features: {
    paginate: false,
    recordCount: false,
    sorting: false
  },
  inputs: {
    queries: $('#search-year')
  }
});
```

Doing it this way also hooks the query into the `pushState` functionality to update the page URL parameters and cache the query result for the browser's forward- and back-buttons, and sets the query event (the JS event that processes the query) to the `inputs.queryEvent` setting (which can also be customized per-input via the `data-dynatable-query-event` attribute). The key-name for the query will be set to the `data-dynatable-query` attribute, the `name` attribute, or the `id` for the input.

Using our own query filters, we may also need something other than text-matching. Perhaps we want a filter which sets a price range. We can add our query input with the `inputs.queries` setting as above, and then define our own query function for that key.

*When using our own query function, the query key must match the name of the query function, rather than the name of a column or record attribute.*

```

$('#search-function-example')
  .bind('dynatable:init', function(e, dynatable) {
    dynatable.queries.functions['max-price'] = function(record, queryValue) {
      return parseFloat(record.price.replace(/,/,'')) <= parseFloat(queryValue);
    };
  })
  .dynatable({
    features: {
      paginate: false,
      recordCount: false,
      sorting: false,
      search: false
    },
    inputs: {
      queries: $('#max-price')
    }
  });

```

By default, when a query is added, dynatable will first look in the `queries.functions` object to find the query function matching the query's key-name. If none is found, it will fall-back to doing a plain-text search on the record attribute matching the query key-name. If that attribute doesn't exist either, then dynatable will throw an error alerting us to add the function.

The query function is called once for each record and should return either `true` or `false`, letting dynatable know if that record matches the query or not.

Max Price: \$

Make	Model	Year	Price
Volkswagen	Jetta Wolfsburg	2008	11,000
Ford	Focus	2013	20,000
Ford	Escape	2001	4,000
Mini	Cooper	2001	8,500
Ford	Focus SVT	2003	9,000

## Paginating

Dynatable also provides pagination by default, by selecting a specific slice of the JSON record collection to render to the page, and adding page selection links to the table, as well as a drop-down allowing the user to select how many records are shown per page.

In other words, dynatable is aware that the currently rendered records in the DOM may only be a subset of the total records.

We can customize the default number of records displayed per page via the `dataset.perPageDefault` configuration setting. And we can customize the per-page options via the `dataset.perPageOptions` configuration setting.

We can also set the page and perPage values via the dynatable API:

```

var dynatable = $('#my-table').data('dynatable');
dynatable.paginationPerPage.set(20); // Show 20 records per page
dynatable.paginationPage.set(5); // Go to page 5
dynatable.process();

```

If `dataset.ajax` is enabled, then the page and per-page parameters are simply passed to the server.

## Record Count

When pagination is enabled, dynatable will also show the currently displayed records and the total number of records in the form:

Showing {x} to {y} out of {z} records

This message can be customized via the `dataset.recordCountText` configuration, and the `params.records` configuration. The text displayed on the table is of the form:

{dataset.recordCountText} {x} to {y} out of {z} {params.records}

Dynatable will also show the queried and total record counts when querying data, in the form:

```
Showing {x} of {y} records (filtered from {z} total records)
```

Or more accurately:

```
{dataset.recordCountText} {x} of {y} {params.records} (filtered from {z} total {params.records})
```

When `dataset.ajax` is enabled, in order for dynatable to display this message, our server must return the number of total records in addition to the sliced record set for the current page. By default, dynatable looks for the total number of records in the `responseJSON.totalRecordCount` attribute.

## PushState

Dynatable uses HTML5's pushState to store operation results (sorting, querying and paginating) and update the browser's URL, so that we may hit the browser's back- and forward-buttons to step through our interactions with the table.

If the resulting data can be stored in the browser's pushState cache, then it will be, and dynatable will simply render the cached data for that step rather than re-running the (potentially complex) operations. If `dataset.ajax` is enabled, then dynatable will render the pushState-cached results rather than re-submitting the AJAX request to the server.

If the resulting dataset for a given operation is too large for the pushState cache, then dynatable will automatically fallback to re-running the operations or re-sending the AJAX request to the server.

PushState works in all modern browsers that support it (<http://caniuse.com/#search=pushstate>). For other browsers (IE9 or earlier), a pushState polyfill such as History.js (<https://github.com/browserstate/history.js>) may be used.

## Processing Indicator

For long-running operations (and for AJAX tables which must request data from the server), dynatable automatically appends a "processing" indicator to the table to let users know something is happening. We can style this indicator however we want. By default, it's just the word "Processing..." overlaid in the center of the table.

We can customize the html content of the processing indicator (including images or gifs), using the `inputs.processingText` configuration.

We can also style the processing indicator overlay and inner block, by attaching styles to the `dynatable-processing` class and the `.dynatable-processing span` CSS selector, respectively.

*Important Things*

E=MC<sup>2</sup>

F=MA

A<sup>2</sup>+B<sup>2</sup>=C<sup>2</sup>

**Important Things**

E=MC<sup>2</sup>

F=MA

A<sup>2</sup>+B<sup>2</sup>=C<sup>2</sup>

Show Standard Processing Indicator

Show Nicer Processing Indicator

To show or hide the processing indicator above, we can call the `dynatable.processingIndicator.show()` and `dynatable.processingIndicator.hide()` functions.

For the nicer example, we just add our own custom markup for the processing indicator, along with some custom CSS.

```
$('#processing-indicator-nice-example').dynatable({
  inputs: {
    processingText: 'Loading '
  }
});
```

```

.dynatable-processing {
  background: #000;
  opacity: 0.6;
  -webkit-border-radius: 4px;
  -moz-border-radius: 4px;
  border-radius: 4px;
}
.dynatable-processing span {
  background: #FFF;
  border: solid 2px #57A957;
  color: #333;
  padding: 25px;
  font-size: 2em;
  box-shadow: 0px 0px 15px rgba(0,0,0,0.5);
}
.dynatable-processing span img {
  vertical-align: middle;
}

```

## Rendering

When rendering JSON data to the page, dynatable passes data through "writers" (you may notice that this is the opposite of the normalization step which runs the DOM elements through "readers").

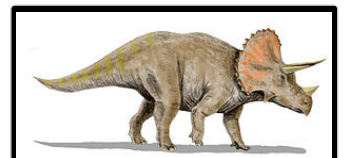
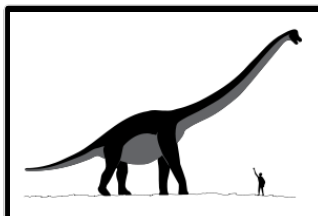
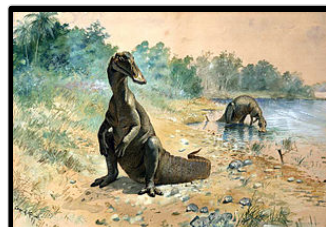
When rendering (and normalizing), dynatable assumes that our container element (on which we called dynatable) contains elements matching `table.bodyRowSelector`, each mapping to one record. By default, dynatable assumes we're rendering to an HTML table, so our `table.bodyRowSelector` is `'tbody tr'`.

To render our records, dynatable will loop through our records, running `writers._rowWriter` on each record to create a collection of DOM elements. The default `writers._rowWriter` creates a table `tr` element and loops through the element attributes (matching our columns) to call `writers._cellWriter` on each.

## A Stylized List

Show:

Search:



Showing 4 to 6 of 6 dinosaurs

Pages: Previous 1 2 Next

\* List of U.S. state dinosaurs from Wikipedia ([http://en.wikipedia.org/wiki/List\\_of\\_U.S.\\_state\\_dinosaurs](http://en.wikipedia.org/wiki/List_of_U.S._state_dinosaurs))

If our container element is a `ul`, like above, we could customize our `rowWriter` as follows:

```

<ul id="ul-example" class="row-fluid">
  <li class="span4" data-color="gray">
    <div class="thumbnail">
      <div class="thumbnail-image">
        
      </div>
      <div class="caption">
        <h3>Stegosaurus armatus</h3>
        <p>State: Colorado</p>
        <p>Year: 1982</p>
        <p><a href="http://en.wikipedia.org/wiki/Stegosaurus" class="btn btn-primary">View</a> <a href="#"
class="btn">View</a></p>
      </div>
    </div>
  </li>
<!-- ... //-->
</ul>

```

```

// Function that renders the list items from our records
function ulWriter(rowIndex, record, columns, cellWriter) {
  var cssClass = "span4", li;
  if (rowIndex % 3 === 0) { cssClass += ' first'; }
  li = '<li class="' + cssClass + '"><div class="thumbnail"><div class="thumbnail-image">' + record.thumbnail + '</div>
<div class="caption">' + record.caption + '</div></div></li>';
  return li;
}

// Function that creates our records from the DOM when the page is loaded
function ulReader(index, li, record) {
  var $li = $(li),
      $caption = $li.find('.caption');
  record.thumbnail = $li.find('.thumbnail-image').html();
  record.caption = $caption.html();
  record.label = $caption.find('h3').text();
  record.description = $caption.find('p').text();
  record.color = $li.data('color');
}

$('#ul-example').dynatable({
  table: {
    bodyRowSelector: 'li'
  },
  dataset: {
    perPageDefault: 3,
    perPageOptions: [3, 6]
  },
  writers: {
    _rowWriter: ulWriter
  },
  readers: {
    _rowReader: ulReader
  },
  params: {
    records: 'kittens'
  }
});

```

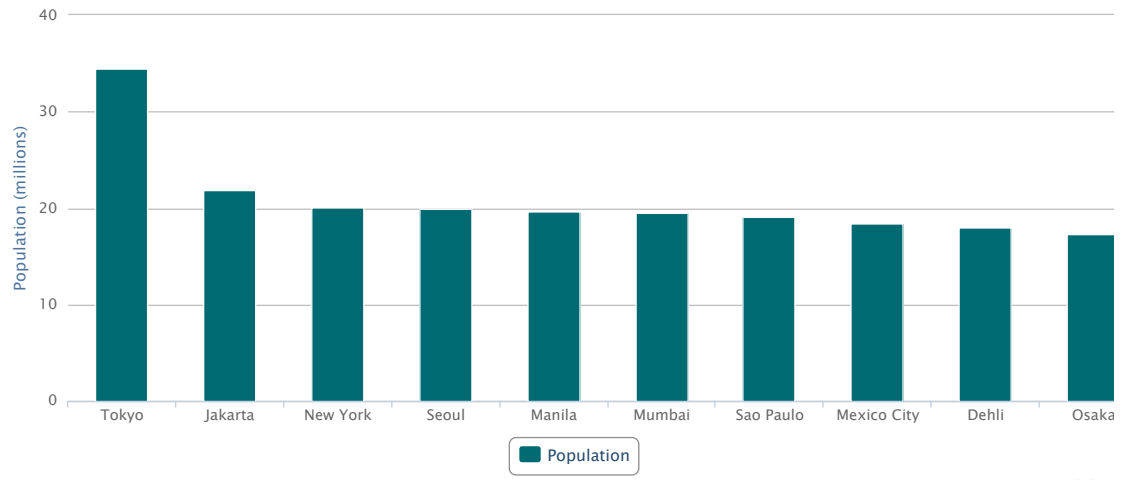
We could have defined our own `writers._cellWriter` as well, defining a custom function for rendering each attribute within the row, but we opted to skip it entirely and to just do everything in the `writers._rowWriter`.

## An Interactive Chart

Show:

Search:

## World's largest cities per 2008



Highchart

Showing 1 to 10 of 20 records

Pages: [Previous](#) [1](#) [2](#) [Next](#)

Show Table to Sort the Chart Series

Our initial data:

```
<div id="chart-example-chart"></div>
<a class="btn primary" id="toggle-chart-table">Show Table to Sort the Chart Series</a>
<table id="chart-example" class="table table-bordered">
  <thead><tr><th>City</th><th>Population</th></tr></thead>
  <tbody>
    <tr><td>Tokyo</td><td>34.4</td></tr>
    <tr><td>Jakarta</td><td>21.8</td></tr>
    <tr><td>New York</td><td>20.1</td></tr>
    <tr><td>Seoul</td><td>20</td></tr>
    <tr><td>Manila</td><td>19.6</td></tr>
    <tr><td>Mumbai</td><td>19.5</td></tr>
    <tr><td>Sao Paulo</td><td>19.1</td></tr>
    <tr><td>Mexico City</td><td>18.4</td></tr>
    <tr><td>Dehli</td><td>18</td></tr>
    <tr><td>Osaka</td><td>17.3</td></tr>
    <tr><td>Cairo</td><td>16.8</td></tr>
    <tr><td>Kolkata</td><td>15</td></tr>
    <tr><td>Los Angeles</td><td>14.7</td></tr>
    <tr><td>Shanghai</td><td>14.5</td></tr>
    <tr><td>Moscow</td><td>13.3</td></tr>
    <tr><td>Beijing</td><td>12.8</td></tr>
    <tr><td>Buenos Aires</td><td>12.4</td></tr>
    <tr><td>Guangzhou</td><td>11.8</td></tr>
    <tr><td>Shenzhen</td><td>11.7</td></tr>
    <tr><td>Istanbul</td><td>11.2</td></tr>
  </tbody>
</table>
```

The JS:

```

(function() {
  var $table = $('#chart-example'), $chart = $('#chart-example-chart'), chart;

  // Create a button to toggle our table's visibility.
  // We could just hide it completely if we don't need it.
  $('#toggle-chart-table').click(function(e) {
    e.preventDefault();
    $table.toggle();
  });

  // Set up our Highcharts chart
  chart = new Highcharts.Chart({
    chart: {
      type: 'column',
      renderTo: 'chart-example-chart'
    },
    title: {
      text: 'World\'s largest cities per 2008'
    },
    yAxis: {
      min: 0,
      title: {
        text: 'Population (millions)'
      }
    },
    series: [{
      name: 'Population',
      color: '#006A72'
    }]
  });

  // Create a function to update the chart with the current working set
  // of records from dynatable, after all operations have been run.
  function updateChart() {
    var dynatable = $table.data('dynatable'), categories = [], values = [];
    $.each(dynatable.settings.dataset.records, function() {
      categories.push(this.city);
      values.push(parseFloat(this.population));
    });

    chart.xAxis[0].setCategories(categories);
    chart.series[0].setData(values);
  };

  // Attach dynatable to our table, hide the table,
  // and trigger our update function whenever we interact with it.
  $table
    .dynatable({
      inputs: {
        queryEvent: 'blur change keyup',
        recordCountTarget: $chart,
        paginationLinkTarget: $chart,
        searchTarget: $chart,
        perPageTarget: $chart
      },
      dataset: {
        perPageOptions: [5, 10, 20],
        sortTypes: {
          'population': 'number'
        }
      }
    })
    .hide()
    .bind('dynatable:afterProcess', updateChart);

  // Run our updateChart function for the first time.
  updateChart();
})();

```

## Configuration

If you want to change any of the following default configuration options globally (for all instances of dynatable within your application), you can call the `$.dynatableSetup()` function to do so:

```

$.dynatableSetup({
  // your global default options here
});

```



For example, this documentation page has `features: { pushState: false}` so as not to fill your browser's pushState queue as you click around through made-up data in the examples (except for the first example, which re-enables it for demo purposes).

The configuration options (with default values) for dynatable are:

```
{
  features: {
    paginate: true,
    sort: true,
    pushState: true,
    search: true,
    recordCount: true,
    perPageSelect: true
  },
  table: {
    defaultColumnIdStyle: 'camelCase',
    columns: null,
    headRowSelector: 'thead tr', // or e.g. tr:first-child
    bodyRowSelector: 'tbody tr',
    headRowClass: null
  },
  inputs: {
    queries: null,
    sorts: null,
    multisort: ['ctrlKey', 'shiftKey', 'metaKey'],
    page: null,
    queryEvent: 'blur change',
    recordCountTarget: null,
    recordCountPlacement: 'after',
    paginationLinkTarget: null,
    paginationLinkPlacement: 'after',
    paginationPrev: 'Previous',
    paginationNext: 'Next',
    paginationGap: [1,2,2,1],
    searchTarget: null,
    searchPlacement: 'before',
    perPageTarget: null,
    perPagePlacement: 'before',
    perPageText: 'Show: ',
    recordCountText: 'Showing ',
    processingText: 'Processing...'
  },
  dataset: {
    ajax: false,
    ajaxUrl: null,
    ajaxCache: null,
    ajaxOnLoad: false,
    ajaxMethod: 'GET',
    ajaxDataType: 'json',
    totalRecordCount: null,
    queries: null,
    queryRecordCount: null,
    page: null,
    perPageDefault: 10,
    perPageOptions: [10,20,50,100],
    sorts: null,
    sortKeys: null,
    sortTypes: {},
    records: null
  },
  // Built-in writer functions,
  // can be overwritten, any additional functions
  // provided in writers will be merged with
  // this default object.
  writers: {
    _rowWriter: defaultRowWriter,
    _cellWriter: defaultCellWriter,
    _attributeWriter: defaultAttributeWriter
  },
  // Built-in reader functions,
  // can be overwritten, any additional functions
  // provided in readers will be merged with
  // this default object.
  readers: {
    _rowReader: null,
    _attributeReader: defaultAttributeReader
  },
  params: {
    dynatable: 'dynatable',
    queries: 'queries',
```

```

    sorts: 'sorts',
    page: 'page',
    perPage: 'perPage',
    offset: 'offset',
    records: 'records',
    record: null,
    queryRecordCount: 'queryRecordCount',
    totalRecordCount: 'totalRecordCount'
  }
}

```

## Data Attributes

In addition to the configuration options directly available above, some properties apply specifically to certain columns or elements. Those can be set using HTML5 data attributes.

Documentation on each data-attribute and what it does coming soon.

### On table column headers

`data-dynatable-column`

`data-dynatable-sorts`

`data-dynatable-no-sort`

### On query inputs

`data-dynatable-query-event`

`data-dynatable-query`

## Event Hooks

Event	Description	Parameters
<code>dynatable:init</code>	Run after dynatable is initialized and setup, right before the initial <code>process()</code> is run.	<code>dynatable</code> (attached dynatable instance object)
<code>dynatable:beforeProcess</code>	Run at the beginning of <code>process()</code> .	<code>data</code> (the data object containing the settings and records for the <code>process()</code> function)
<code>dynatable:ajax:success</code>	Run only if the dynatable instance has <code>dataset.ajax=true</code> , when the AJAX request returns successfully during the <code>process()</code> function.	<code>response</code> (the jqXHR response object)
<code>dynatable:afterProcess</code>	Run at the end of the <code>process()</code> function.	<code>data</code> (the data object containing the settings and records for the <code>process()</code> function)
<code>dynatable:beforeUpdate</code>	Run right before the DOM is updated with the current record set.	<code>\$rows</code> (the set of DOM rows about to be written to the DOM)
<code>dynatable:afterUpdate</code>	Run right after the DOM is updated with the current record set.	<code>\$rows</code> (the set of DOM rows just written to the DOM)
<code>dynatable:push</code>	Run when pushState data is pushed to the window.	<code>data</code> (the data object containing the settings and records to be cached in the pushState cache)

## API

You can interface directly with the dynatable API for finer grained control and greater customization. The internal API is divided into namespaces. To use the API, just call the namespaced function on the dynatable object (stored in the `data['dynatable']` attribute of the element on which dynatable was called).

```
var dynatable = $('#my-table').data('dynatable');
```

For example, to update the dom with the current record set:

```
dynatable.dom.update();
```

Since dynatable is still pre-version-one, the API is still in flux and may change. Below is a list of the current API functions and arguments (if any).

## dom

update

## domColumns

getFromTable

add [\$column, position, skipAppend, skipUpdate]

remove [columnIndexOrId]

removeFromTable [columnId]

removeFromArray [index]

generate [\$cell]

attachGeneratedAttributes

## records

updateFromJson [data]

sort

paginate

resetOriginal

pageBounds

getFromTable

count

## recordsCount

create

attach

## processingIndicator

create

position

attach

show

hide

## state

push [data]

pop [event]

## sorts

add [attr, direction]

remove [attr]

clear

guessType [a, b, attr]

functions (object)

## sortsHeaders

create [cell]

attach

attachOne [cell]

appendArrowUp [\$link]

appendArrowDown [\$link]

removeArrow [\$link]

removeAllArrows

toggleSort [event, \$link, column]

sortedByColumn [\$link, column]

sortedByColumnValue [column]

## queries

add [name, value]

remove [name]

run

runSearch [query]

setupInputs

functions (object)

## inputSearch

create

attach

## paginationPage

set [page]

## paginationPerPage

create

attach

set [number]

## paginationLinks

create

attach

---

Congratulations, you've reached the end of the documentation!

Take these links. They will help you in your journey:

---

## Project Resources

- Report bug or request feature (<https://github.com/alfajango/jquery-dynatable/issues>)